Universidad Carlos III de Madrid
Escuela Politécnica Superior

PROYECTO FIN DE CARRERA

Ingeniería Informática (2º Ciclo)

EVOLUCIÓN DE PROYECCIONES LINEALES PARA APRENDIZAJE AUTOMÁTICO

Autor
Miguel García-Serrano González

Directores
José María Valls Ferrán
Ricardo Aler Mur

Leganés, Julio de 2009

ÍNDICE

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

ÍNDICE DE ECUACIONES

1. Introducción

En este capítulo se resumen los objetivos que se persiguen con el presente proyecto. Además, se describe el contenido de cada capítulo de la memoria.

1.1 Resumen y objetivos del proyecto

En este proyecto se van a utilizar técnicas evolutivas para mejorar los resultados en tareas de clasificación y regresión de aprendizaje automático. Un algoritmo genético es un tipo especial de algoritmo que se inspira en la evolución biológica para encontrar soluciones a problemas y puede utilizarse para optimizar funciones. Por otro lado, muchos de los algoritmos de aprendizaje automático pueden aprender de un conjunto de datos clasificando a los nuevos datos teniendo en cuenta cómo están clasificados datos similares. Se puede mejorar la precisión de los algoritmos de aprendizaje (clasificación o regresión) transformando el espacio en el que se representan los datos. Esta transformación puede llevarse a cabo de manera lineal, multiplicando los datos por una matriz. Una transformación lineal es una función entre dos espacios vectoriales que preserva las operaciones de suma y producto por un escalar.

Por tanto, las transformaciones que se van a considerar consisten en multiplicar los datos originales por una matriz. Si llamamos **x** al vector que contiene los atributos de entrada de los datos representados en el espacio original, **x'**=**x**·M representará los datos transformados. En esta expresión, M es la matriz que realiza la transformación y podrá ser:

- Una matriz cuadrada, en cuyo caso las dimensiones de los datos transformados **x'** es la misma que la de los originales **x**.
- Una matriz rectangular de tamaño *nxm*, donde n son las dimensiones del espacio original y m las del proyectado. Si $m<n$ reduciremos la dimensionalidad y si $m>n$ la aumentaremos.

También se considerarán tres tipos de matrices: diagonales, simétricas y completas.

- Las matrices diagonales sólo tienen valores en la diagonal y ceros en el resto.
- Las matrices simétricas tienen los mismos valores en el triángulo inferior izquierdo que en el triángulo superior derecho.
- Las matrices completas tienen todos sus valores.

El uso de muchos parámetros (matriz completa) puede inducir sobreadaptación, mientras que el uso de pocos (matriz diagonal) puede llevar a subadaptación. La matriz más apropiada dependerá del problema y se determinará mediante experimentación.

A continuación, se ilustran diferentes ejemplos de cómo los espacios de datos pueden ser transformados por diferentes matrices. En las ilustraciones 1.1 y 1.2 pueden verse matrices que provocan rotaciones de los datos:

$$M_1 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

Ilustración 1.1 – Rotación de 90º en el sentido de las agujas del reloj

$$M_2 = \begin{pmatrix} \cos\theta & -sen\theta \\ sen\theta & \cos\theta \end{pmatrix}$$

Ilustración 1.2 – Rotación de θ grados en sentido contrario a las agujas del reloj

En la ilustración 1.3 se muestra la matriz con la que puede conseguirse reflejar los datos en el eje X.

$$M_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Ilustración 1.3 – Reflexión en el eje X

La matriz que aparece en la ilustración 1.4 escala los datos en todas direcciones (en este caso duplicando sus valores). En general, el reescalado es el efecto de todas las matrices diagonales.

$$M_4 = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

Ilustración 1.4 – Escalado en todas direcciones

A modo de ejemplo, las ilustraciones 1.5 y 1.6 muestran el escalado de un conjunto de datos transformado con la matriz M_4:

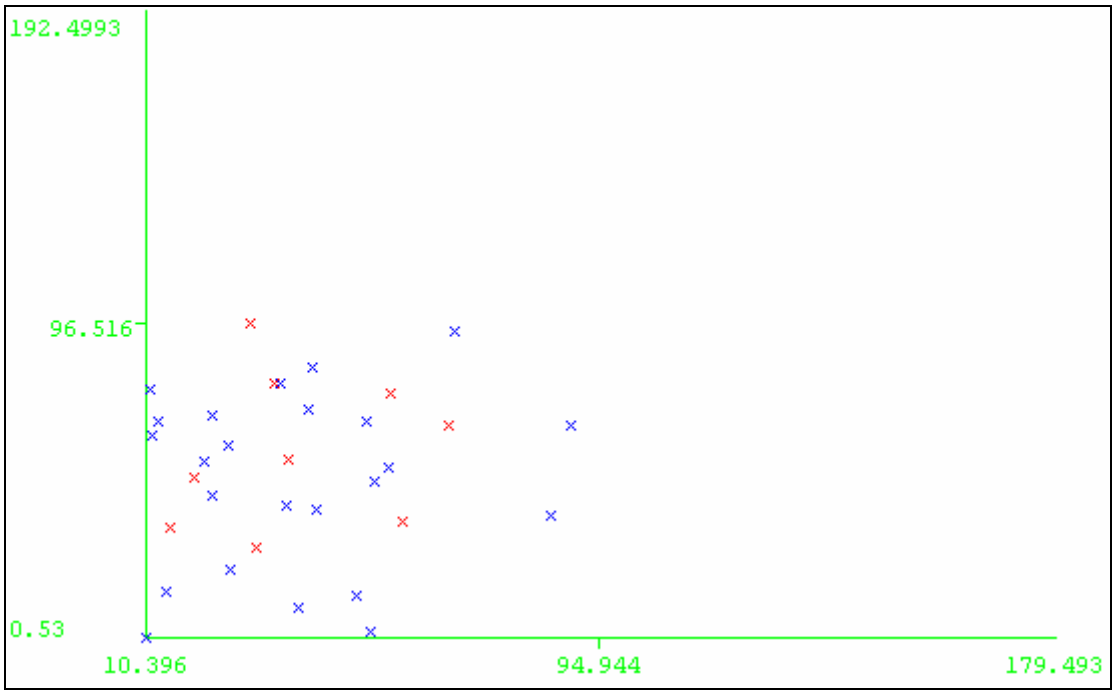

Ilustración 1.5 – Conjunto de datos original

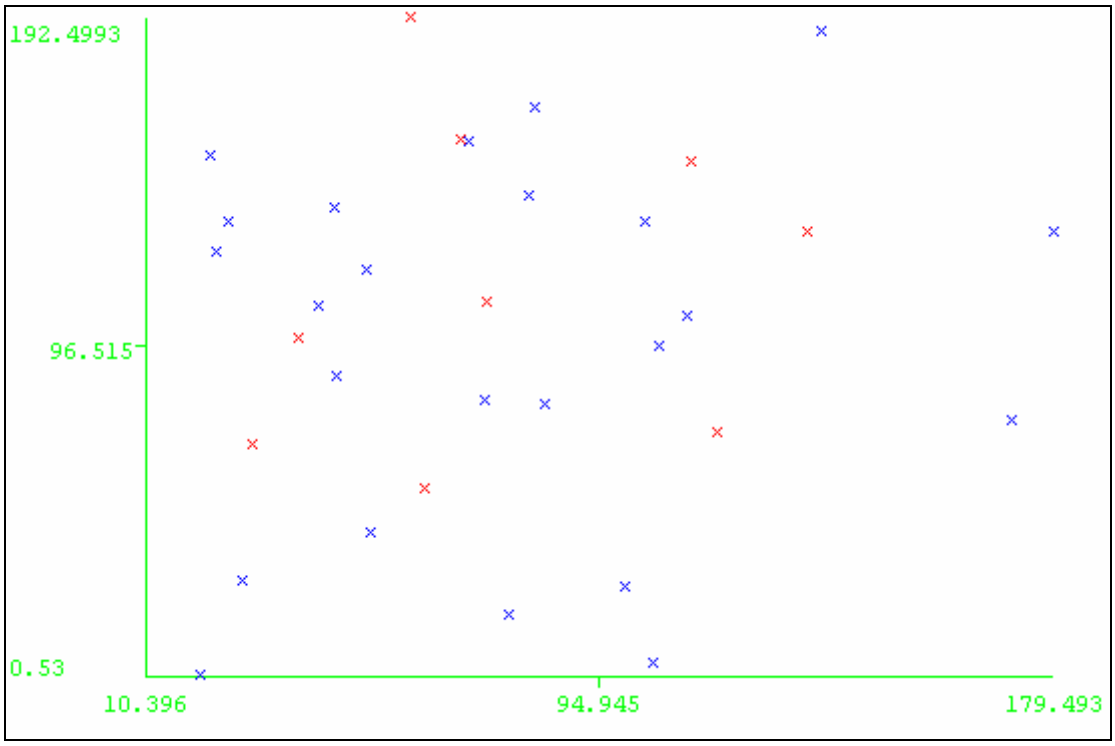

Ilustración 1.6 – Conjunto de datos transformado con la matriz M_4

El objetivo principal del proyecto es utilizar técnicas evolutivas para encontrar la matriz M que optimice el porcentaje de aciertos en problemas de clasificación (o el error cuadrático en problemas de regresión). Es decir, se espera que para un algoritmo de clasificación dado, que denominaremos algoritmo base (por ejemplo, *J48*), el porcentaje de aciertos en test para el conjunto de datos transformados **x'** sea mas alto que para el conjunto original **x**. La matriz M no es más que un conjunto de valores reales, y se sabe

que las técnicas evolutivas son apropiadas para optimizar cromosomas compuestos de reales. La función a optimizar (la función de fitness) será el porcentaje de aciertos obtenido al transformar los datos de entrenamiento con el individuo (una matriz M concreta).

En el presente proyecto no sólo se quieren implementar y probar las ideas anteriores, sino integrarlas dentro del sistema de aprendizaje automático *Weka*, para que la transformación de datos pueda ser utilizada con cualquiera de los algoritmos y filtros proporcionados por dicha herramienta. De hecho, nuestra proyección evolutiva será implementada como un meta-algoritmo, que use otro algoritmo base, que será el beneficiario de la proyección. A pesar de que *Weka* permite la integración de nuevos algoritmos, este proceso está lejos de ser trivial, por lo que la inclusión de la técnica evolutiva dentro de *Weka* es una parte importante de este proyecto y ha consumido una buena parte del tiempo del mismo.

Además, el meta-algoritmo permite una exhaustiva configuración de los más importantes parámetros que intervienen en un algoritmo evolutivo en general (tamaño de la población, constante de decremento…) y, en este en particular (tipo de matriz (completa, simétrica o diagonal), exponente…).

Por último, también se llevará a cabo una validación experimental del sistema desarrollado.

1.2 Estructura del proyecto

En este apartado se detalla el contenido de cada capítulo de la memoria del presente proyecto:

- ***Capítulo 2. Estado del arte.***

  En este apartado se detallarán los problemas existentes en la actualidad asociados al aprendizaje automático y que el algoritmo desarrollado en este proyecto es capaz de tratar.

  Además, se ilustrarán para cada uno de los problemas los más significativos intentos que la comunidad científica ha llevado a cabo en los últimos años para tratar de hacer frente a estos problemas.

- ***Capítulo 3. Fundamentos del aprendizaje automático.***

  En este capítulo se expondrán los fundamentos del aprendizaje automático, en cuya fundamentación teórica se basa el presente proyecto. En primer lugar se realizará una introducción en la que se definirá y se describirá su historia, posteriormente se explicarán los tipos de aprendizaje automático y los algoritmos existentes. Por último, se describirá qué es la minería de datos y se citarán diferentes aplicaciones del aprendizaje automático.

- ***Capítulo 4. Fundamentos de la computación evolutiva.***

  En este capítulo se expondrán los fundamentos de la computación evolutiva, una de cuyas aplicaciones es el presente proyecto. En primer lugar se realizará una introducción sobre sus fundamentos e historia, posteriormente se explicarán los tres paradigmas en que se compone la misma (algoritmos genéticos, estrategias evolutivas y programación genética) y, por último, se expondrán las críticas que suscita/ha suscitado.

- ***Capítulo 5. Weka.***

  En este apartado se hará una introducción al software libre *Weka* y se detallará el formato de los ficheros de entrada que acepta. Posteriormente, se detallará su funcionalidad para cada una de sus aplicaciones: explorador, experimentador, flujo de conocimiento y consola. Por último, se detalla brevemente la estructura de *Weka*, indicando cada uno de sus paquetes y describiendo sus más importantes clases, para acabar explicando cómo se integran nuevos algoritmos en *Weka*.

- ***Capítulo 6. Descripción del trabajo realizado.***

  En este capítulo se describirá qué desarrollo se ha llevado a cabo en este proyecto. En primer lugar, se indicará qué se pretende conseguir y con qué herramientas/tecnologías va a ser desarrollado. Posteriormente, se indicará la integración en *Weka* y su implementación a alto nivel. En último lugar se explicarán qué ficheros de salida genera el algoritmo desarrollado y se ilustrará un ejemplo de ejecución.

- ***Capítulo 7. Experimentación.***

  En este capítulo se lleva a cabo la experimentación con el meta-algoritmo desarrollado. Se llevarán a cabo cuatro experimentos, tres de ellos con dominios sintéticos especialmente diseñados para poner a pruebas a *PMatrix*. En cada uno de ellos se describirá el conjunto de datos empleado, se experimentará sin y con *PMatrix* para comparar la diferencia y se describirán las conclusiones de dicho experimento. Por último, se comentan las conclusiones globales de la experimentación.

- ***Capítulo 8. Conclusiones y líneas futuras.***

  En este capítulo se hará un breve repaso a las conclusiones sobre el trabajo desarrollado y qué mejoras tienen cabida en el mismo.

2. Estado del arte

En este apartado se detallarán los problemas existentes en la actualidad asociados al aprendizaje automático y que el algoritmo desarrollado en este proyecto es capaz de tratar.

Además, se ilustrarán para cada uno de los problemas los más significativos (de los poco numerosos) intentos que la comunidad científica ha llevado a cabo en los últimos años para tratar de hacer frente a estos problemas.

2.1 Problemas actuales

A pesar de la importancia que el aprendizaje automático ha cobrado en la actualidad, el aprendizaje de grandes conjuntos de datos es un problema no abordado de una manera completa.

Se ha comprobado que pueden aprenderse conceptos de conjuntos de datos sorprendentemente grandes, sin embargo, descubrirlos puede no ser abordable por problemas de tiempo y espacio. La complejidad computacional asociada a los algoritmos de aprendizaje puede ser tal que imposibilite el aprendizaje en muchos conjuntos de datos.

En la actualidad, la sociedad de la información, genera datos a un ritmo incesante y cada vez mayor. No es de extrañar que el disponer de un método eficaz y eficiente en el tratamiento/aprendizaje de esos datos resulte vital. No es necesario señalar los beneficios que conllevaría que cada empresa/organización pudiera aprender mejor sus propios datos.

A modo de resumen, la reducción de la dimensionalidad es un problema fundamental en el proceso previo al aprendizaje propiamente dicho (de cualquier tipo) y en la visualización científica. Además, está muy ligada a problemas de candente actualidad, entre los que destacan el reconocimiento del lenguaje, de imágenes y de voz. En términos generales, la reducción de la dimensionalidad es beneficiosa, en general, por los siguientes motivos:

- Mejora la predicción.
- Aumenta la eficiencia del aprendizaje.
- Proporciona modelos más rápidamente y demandando, posiblemente, menos información que la proporcionada originalmente[1].
- Posibilita un mayor entendimiento del problema al proporcionarse modelos en los que menos atributos están involucrados.

[1] Hay que tener en cuenta que la obtención y/o cálculo de los valores de algunos atributos puede ser lenta y/o costosa.

De hecho, existen numerosos algoritmos cuyo único objetivo es descubrir representaciones de pocas dimensiones de conjuntos de datos de alta dimensionalidad, tal es el caso del algoritmo de aprendizaje no supervisado *LLE*.

Sin embargo, a pesar de hablar continuamente de grandes conjuntos de datos, quizás no siempre resulte útil una reducción de su dimensión, puesto que el conjunto de datos podría tener una pequeña dimensionalidad. El aumento de la dimensionalidad podría ser útil para el descubrimiento de nuevos atributos/características que podrían resultar de gran utilidad y facilitar enormemente el aprendizaje. Por ejemplo, un algoritmo de aprendizaje produciría resultados más fácilmente exitosos trabajando con la característica densidad de población, que trabajando con las características población y superficie.

Por último, en muchas ocasiones puede resultar útil una mera transformación del espacio, de modo que el espacio transformado resulte más fácil de aprender (por un algoritmo determinado). La transformación del espacio es equivalente a utilizar el espacio original utilizando una función de distancia diferente, con la ventaja de que, realizando transformaciones, el espacio original puede ser aprendido por todo tipo de algoritmos, y no solamente por aquellos que utilicen funciones de distancia en su aprendizaje.

En la ilustración 2.1 se muestra cómo un algoritmo de clasificación lineal (por ejemplo, *J48*) podría mejorar el porcentaje de aciertos, a la par que simplificar su modelo si previamente se lleva a cabo una transformación (φ) de los datos:

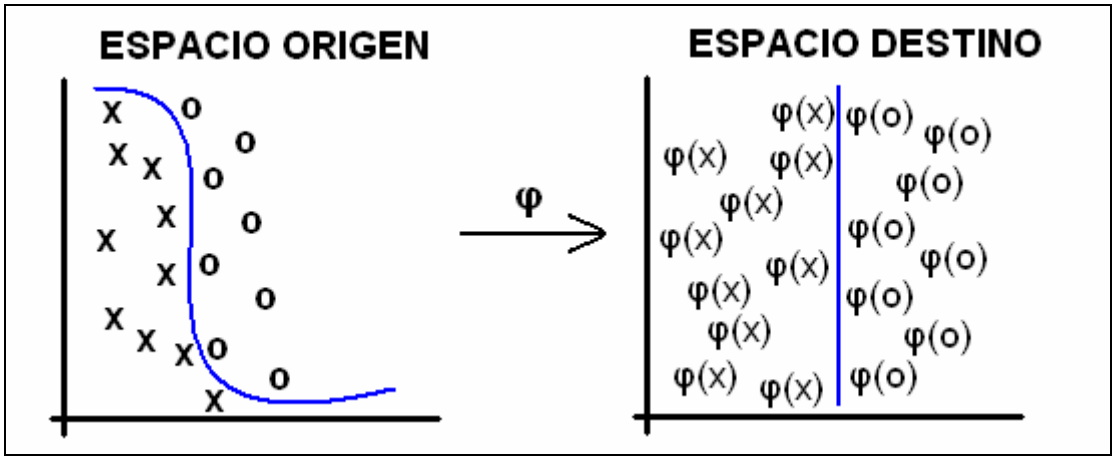

Ilustración 2.1 – Ejemplo de transformación de los datos

2.2 Técnicas estándar

Una de las técnicas más empleadas para reducir la dimensionalidad es la selección de atributos, cuya misión es la eliminación de atributos irrelevantes, redundantes…

PCA (*Principal Component Analysis*, Análisis de Componentes Principales en castellano) es una técnica inventada por Karl Pearson en 1901 [8]. PCA es un procedimiento matemático que transforma un conjunto de variables (atributos) correladas en un conjunto menor de variables no correladas (llamadas componentes principales). La primera variable intenta explicar la mayor cantidad de datos (tanta variabilidad) como sea posible, y cada componente siguiente se centra en explicar la variabilidad del resto de datos.

LLE (*Locally Linear Embedding*) es un algoritmo de aprendizaje no supervisado que permite descubrir representaciones de datos con una gran dimensionalidad en una dimensión (más) baja. *LLE* consigue su objetivo explotando simetrías locales de reconstrucciones lineales sin que sus optimizaciones impliquen mínimos locales.

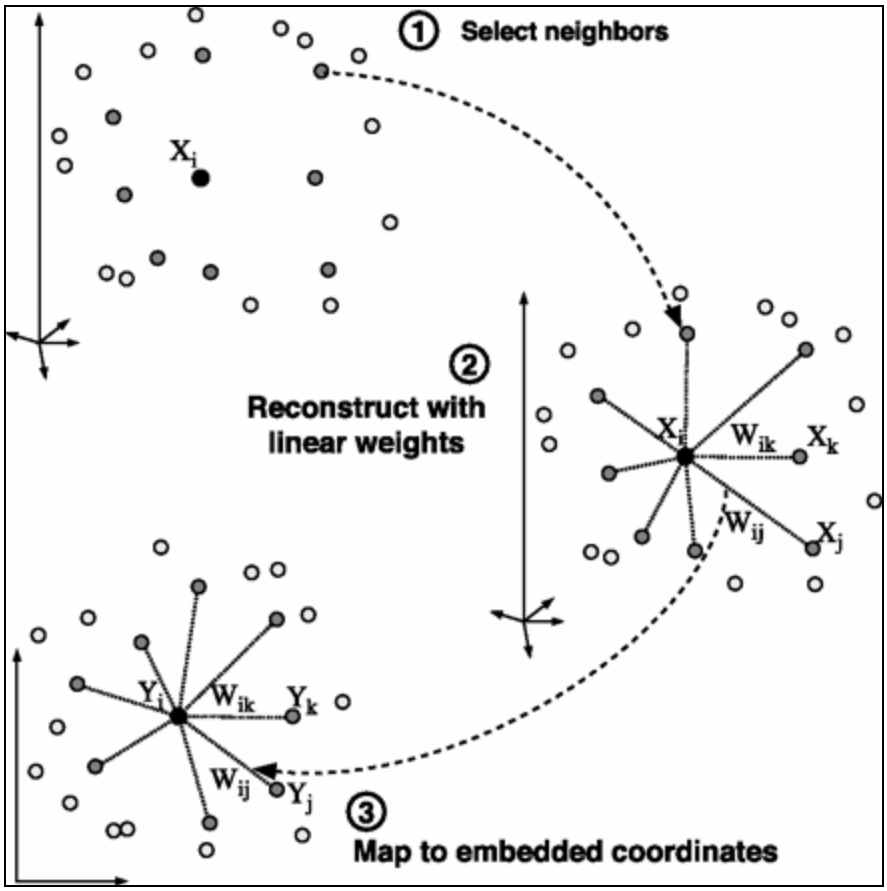

Ilustración 2.2 – Pasos del algoritmo LLE

Otras técnicas importantes que pueden utilizarse para reducir la dimensionalidad son:

- Escalamiento multidimensional (Multidimensional Scaling, MDS), mapeado de Sammon…
  Transformada de Karhunen Loeve (Karhunen Loeve transform, KLT)
- Descomposición en valores singulares (singular value decomposition, SVD)
- Mapas topográficos
  Mapas autoorganizativos (Self-organized maps, SOM)
- Métodos basados en redes neuronales
- Isomap

El paquete de software *Weka* dispone de un filtro para aplicar a los conjuntos de datos que le son cargados:

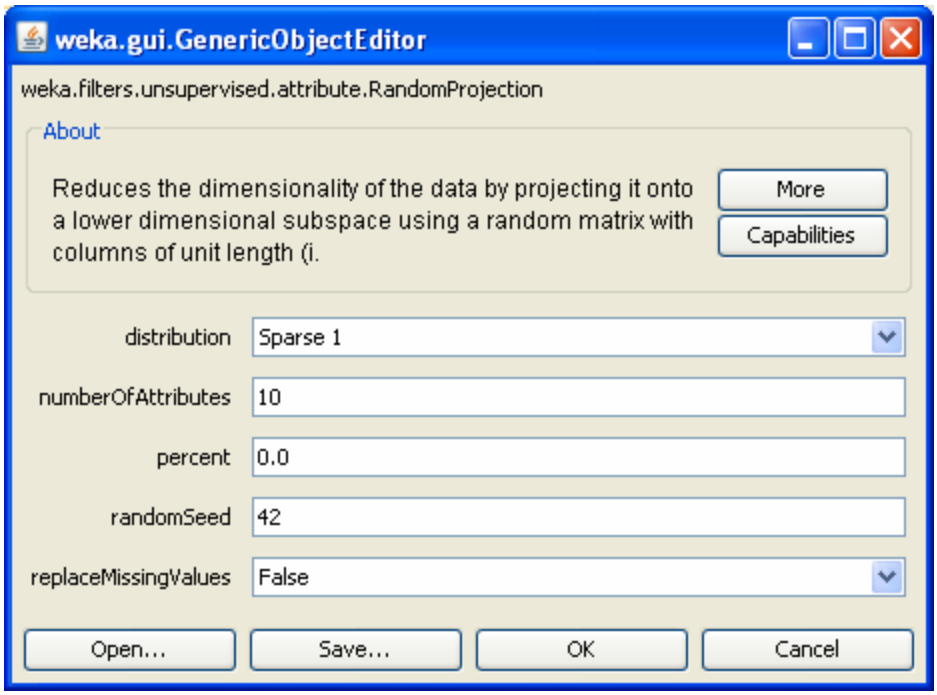

Ilustración 2.3 – Filtro *RandomProjection* deWeka

Este filtro no supervisado de atributo permite realizar transformaciones de los datos haciendo uso de una matriz generada de manera aleatoria. Asimismo, si el usuario establece para el parámetro número de atributos un valor diferente al número de atributos del espacio origen, dicha matriz provocará que el espacio transformado tenga una dimensionalidad diferente.

2.3 Investigaciones actuales

Gran parte de las investigaciones realizadas hasta la fecha que pretenden conseguir algunos de los objetivos logrados por *PMatrix*, se centran en el uso de proyecciones aleatorias. A modo de ejemplo, se presentan algunos trabajos aparecidos en las últimas fechas:

- Ran Gilad-Bachrach, en su artículo [*9*] utiliza las proyecciones aleatorias conjuntamente con un meta-algoritmo que proyecta a una dimensión menor los datos de entrada que va recibiendo. Posteriormente demuestra su efectividad con el algoritmo de aprendizaje online *Bayes Point Machine*.

- Mingrui Wu del Instituto Max Planck desarrolla un nuevo algoritmo [*10*] basado en proyecciones locales, que conduce a un error de estimación local mínimo. Posteriormente, comparan dicho algoritmo con PCA y el novedoso algoritmo LPP.

- Yu-En Lu, Pietro Li'o y Steven Hand desarrollan un nuevo algoritmo [*11*], llamado *Beta Random Projection*, basado en proyecciones aleatorias, inspirado por el uso de distribuciones de probabilidad simétricas. Posteriormente lo comparan con otro algoritmo basado en proyecciones aleatorias, *SVD*.

- Andreas Buja y George W. Furnas, construyen vistas [*12*] de conjuntos de datos de alta dimensionalidad a través de secciones y proyecciones. A través de proyecciones consiguen visualizar aspectos de la estructura que sólo son reconocibles cuando se trabaja en bajas dimensiones.

José M. Valls, Ricardo Aler y Óscar Fernández [*16*], utilizan un algoritmo genético para evolucionar matrices de distancias euclídeas generalizadas con el objetivo de utilizarlas como función de activación de redes neuronales de base radial (*RBNN*), de tal forma que el fitness de cada matriz está condicionado por la predicción conseguida por la misma. De esta manera, se superar algunos problemas, tales como que no todos los atributos son igualmente relevantes y que su relevancia está directamente relacionada con el sistema de aprendizaje utilizado.

Kilian Q. Weinberger y Lawrence K. Saul [18], conocedores de que *KNN* trabaja con distancias euclídeas simples y estas tienen el inconveniente de que ignoran cualquier regularidad estadística[2] que puede darse en los conjuntos de datos, son conscientes de que pueden mejorarse los resultados de *KNN* modificando dichas funciones de distancia. Para ello, y para mejorar trabajos previos, aprenden no una, sino varias métricas de distancia (*Mahalanobis*), cada una asociada a una clase y/o región del espacio diferente (todas ellas entrenadas simultáneamente tratando de minimizar una

[2] Los datos pueden ser muy diferentes entre sí, por ejemplo, dado un conjunto de datos con caras de personas junto con los atributos edad y sexo, se trata de un problema muy diferente el clasificar las caras por edad y sexo, que clasificar la edad o el sexo por el resto de atributos.

simple función de pérdida). Por tanto, el algoritmo que desarrollan intenta optimizar el aprendizaje realizando una transformación del espacio origen (antes de llevar a cabo la clasificación con KNN) empleando, para ello, distancia euclídeas.

2.4 Conclusiones

A pesar de la gran cantidad de artículos que pueden encontrarse con las etiquetas *Random Projections* (proyecciones aleatorias) o *Dimensionality Reduction* (reducción de la dimensionalidad), poco se ha hecho en relación con un procedimiento automático o semiautomático que transforme eficientemente conjuntos de datos. Es decir, la práctica totalidad de los artículos donde se utilizan proyecciones aleatorias utilizan matrices generadas de manera aleatoria y, en los casos en que se empleaban técnicas evolutivas (automáticas) para transformar espacios de datos, ésto se llevaba a cabo con funciones de distancia.

El algoritmo *PMatrix* supone una extensión de las ideas y trabajos presentados en este apartado, pues realiza transformaciones con matrices y no con funciones de distancia, de esta manera, *PMatrix* puede ser empleado con cualquier clasificador y no solamente con aquellos que trabajen con funciones de distancia. Además, se facilita el trabajo con el clasificador base ya que, al estar integrado en *Weka*, el clasificador base puede ser cualquiera de los algoritmos de clasificación implementados en dicho software, sin olvidar que su número se incrementa con el tiempo.

3. Fundamentos del Aprendizaje Automático

En este capítulo se expondrán los fundamentos del aprendizaje automático, en cuya fundamentación teórica se basa el presente proyecto. En primer lugar se realizará una introducción en la que se definirá y se describirá su historia, posteriormente se explicarán los tipos de aprendizaje automático y los algoritmos existentes. Por último, se describirá qué es la minería de datos y se citarán diferentes aplicaciones del aprendizaje automático.

3.1 Introducción

De acuerdo con el diccionario de la Real Academia Española, aprender es adquirir conocimiento de algo por medio del estudio o la experiencia.

El aprendizaje automático es una rama de la inteligencia artificial cuyo objetivo es que las computadoras sean capaces de aprender. En realidad, se trata de que los programas de ordenador sean capaces de inducir conocimiento de cierta información, generalmente conjuntos de ejemplos, que utilizarán posteriormente a su favor.

3.1.1 Aprendizaje animal

Un animal se dice que aprende cuando se adapta a los estímulos que recibe del entorno. Es decir, se puede hablar de aprendizaje cuando recibe información del entorno y la almacena, durante un tiempo relativamente largo, con el objetivo de sacar provecho de ella cuando se le presenten situaciones similares en un futuro.

El aprendizaje puede también considerarse desde un punto de vista biológico. Es decir, el aprendizaje sería un proceso adaptativo en el que se van manifestando una serie de comportamientos e instintos en base a determinados estímulos.

Por último, lo que un animal puede aprender está muy condicionado a su sistema sensorial, en este sentido, sería muy diferente el aprendizaje que se daría entre dos especies idénticas en las que una de las cuales no dispone del sentido de la vista.

Una creencia común es que el aprendizaje sólo se da en seres muy evolucionados, nada más lejos de la realidad.

El condicionamiento es un tipo de aprendizaje, puesto que provoca cambios de comportamiento sobre un animal. Existen los siguientes tipos de condicionamiento:

- Condicionamiento de habituación: El animal tiende a responder con menor intensidad ante un estímulo irrelevante. De esta manera, el animal ahorra tiempo y energía al no reaccionar ante estímulos que no le provocan efecto alguno.

- Condicionamiento clásico: El animal desarrolla una respuesta ante un estímulo neutro, pero que anuncia la llegada de uno que no lo es.
- Condicionamiento de defensa: Permite reaccionar al animal ante pistas que puedan llevarle a algún peligro.
- Condicionamiento de reproducción, condicionamiento operante y condicionamiento asociativo son otros tipos de condicionamiento.

En resumen, el aprendizaje es un cambio de comportamiento, que se da a través de la experiencia e implica una interacción del aprendiz con su entorno.

3.1.2 Aprendizaje automático

Como se comentó en la introducción, el aprendizaje automático es una rama de la inteligencia artificial cuyo objetivo es que las computadoras sean capaces de aprender. En realidad, se trata de que los programas de ordenador sean capaces de inducir conocimiento de cierta información, generalmente conjuntos de ejemplos, que utilizarán posteriormente a su favor. Por ejemplo, en un juego por computador un agente aprende si es capaz de ajustar su estrategia para adaptarse a diferentes oponentes.

Visto así, el aprendizaje automático puede considerarse compuesto por dos fases: una primera en la que elige/observa las características más relevantes de un evento y otra, en la que compara estas características y las adapta a su modelo.

El algoritmo de aprendizaje automático, independientemente del tipo de aprendizaje que lleve a cabo, acepta una información como entrada y genera un modelo que intenta explicar los datos de entrada; pudiendo estar representado dicho modelo, por ejemplo, mediante conjuntos de reglas.

3.1.3 Historia

En la década de 1950, Arthur Samuel construyó programas para jugar a las damas. Samuel fue el primero en utilizar métodos de búsqueda heurística. En 1959 Oliver Selfridge propuso la arquitectura *Pandemonium*, una técnica conexionista primitiva que se basaba en la suposición errónea de que el cerebro está compuesto por múltiples demonios (o procesos activos), cada uno de los cuales es responsable de una tarea determinada.

En 1969, Marvin Minsky y Seymour Papera publicaron *Perceptrons* [*13*], la obra fundacional de las redes de neuronas artificiales. En esta década también se publicaron los primeros trabajos de reconocimiento de patrones.

En la década de los 70, Ross Quinlan desarrolla el algoritmo *ID3*. Se desarrollan los primeros sistemas expertos (*Mycin*), se redescubren leyes numéricas físicas con *BACON*

y *AM* redescubría leyes matemáticas. Además, el algoritmo *AQ* de aprendizaje de reglas fue desarrollado por Riszard S. Michalsky.

En la década de los 80 se desarrollan algoritmos avanzados basados en reglas y árboles de decisión y se desarrolla el algoritmo *EBL* (*Explanation-based Learning*). Las redes de neuronas artificiales cobran un nuevo impulso con la publicación del algoritmo *backpropagation*, surgen arquitecturas cognitivas y se aplica aprendizaje en la resolución de problemas de planificación.

En la década de 1990, surge la minería de datos, la minería de textos, las primeras aplicaciones web y agentes software adaptativos y los primeros meta-algoritmos: *bagging*, *boosting* y *stacking*. Además, se descubren dos paradigmas de aprendizaje, el aprendizaje por refuerzo y la programación lógico inductiva.

En la década actual, aparecen las máquinas de vectores de soporte, los modelos gráficos… Además, el aprendizaje automático se aplica a todo tipo de aplicaciones informáticas (compiladores, detectores de spam, antivirus…). Además, es empleada en robótica y nuestros asistentes personales aprenden para hacernos más cómoda la vida.

Algunas de las disciplinas que contribuyeron al desarrollo del aprendizaje automático fueron:

- **Inteligencia Artificial**: Se han empleado técnicas de búsqueda, de representación de conocimiento…
- **Probabilidades y Estadística**: El teorema de Bayes es la base de numerosos algoritmos en aprendizaje automático. Además, en muchos casos se utilizan intervalos de confianza, cálculo de varianzas…
- **Teoría de Control**: Esta disciplina tuvo que anticiparse al aprendizaje automático porque uno de sus principales problemas era la necesidad de aprender a predecir el próximo estado de un proceso al que controlan.
- **Teoría de la Información**: La medida de la entropía y de contenido de información son utilizadas por numerosos algoritmos.

3.2 Tipos de aprendizaje

Según el tipo de selección y adaptación que un sistema de aprendizaje automático hace de la información que recibe (o se le proporciona), pueden distinguirse varios tipos de aprendizaje, entre los más importantes se encuentran: aprendizaje inductivo, aprendizaje deductivo, aprendizaje supervisado y no supervisado, aprendizaje por refuerzo y aprendizaje conexionista.

3.2.1 Aprendizaje inductivo y deductivo

El aprendizaje inductivo consiste en la adquisición de nuevos conocimientos basado en el descubrimiento de leyes generales o conceptos a partir de un limitado número de ejemplos, basándose en el razonamiento inductivo. Este tipo de aprendizaje es el que utiliza en gran medida el ser humano.

El conocimiento adquirido en aprendizaje inductivo está basado en las posibles similitudes que se dan en los datos. En resumen, este tipo de aprendizaje se basa en una generalización: acepta unos datos de entrada específicos (subconjuntos de todas las posibles situaciones) y produce unos datos de salida generales; dicha salida es un modelo que se aplicará a todos los ejemplos, conocidos o no.

Por el contrario, el aprendizaje deductivo se basa en una deducción. Acepta como datos de entrada un modelo o conjunto de reglas aplicables a todos los ejemplos y produce como salida un conjunto de reglas específicas que pueden ser aplicadas solamente a aquellos ejemplos que cumplan unas determinadas condiciones.

3.2.2 Aprendizaje por refuerzo

El aprendizaje por refuerzo tiene su fundamentación en la psicología experimental, y concretamente en los experimentos realizados por el fisiólogo ruso Iván Pávlov. La idea general es reforzar los comportamientos apropiados y castigar los inapropiados.

En el aprendizaje por refuerzo no hay fuente de información, es decir, no hay datos de entrada. El sistema lleva a cabo una autoexploración infiriendo reglas útiles mediante un proceso iterativo de prueba y error con el fin de tomar una decisión a futuro. Este aprendizaje es considerado aprendizaje supervisado (ver apartado 3.2.3).

En ocasiones, el aprendizaje por prueba y error puede requerir un gran número de repeticiones, por tanto, limita este tipo de aprendizaje a ser utilizado en no muchas aplicaciones prácticas.

El aprendiz (agente, robot…) modela el mundo con estados y acciones, de tal forma que las acciones le llevan de un estado a otro y el objetivo es aprender cuál es la mejor

acción a realizar en cada estado. Un ejemplo de estado puede ser la posición de un robot en una casa y un ejemplo de acción, el de moverse hacia delante.

El esquema básico de comportamiento puede observarse en la ilustración 3.2:

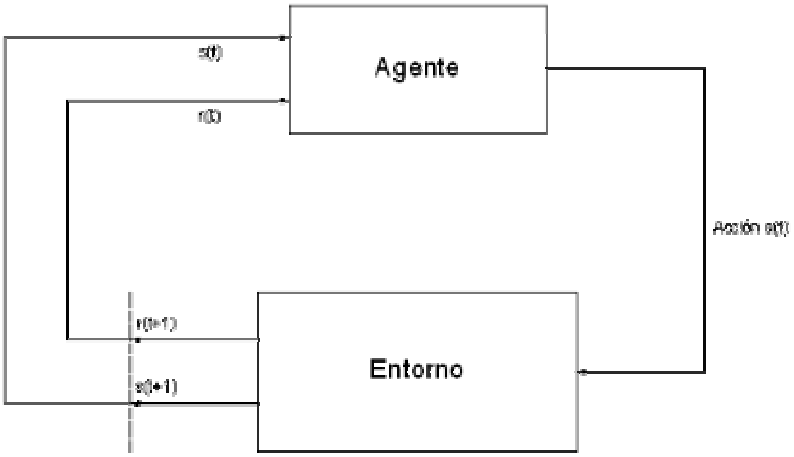

Ilustración 3.1 – Esquema básico en aprendizaje por refuerzo

El aprendizaje por refuerzo es utilizado habitualmente en los siguientes tipos de dominios/problemas:

- Juegos de estrategia: En juegos de estrategias, tales como ajedrez, damas… puede ser útil (aunque intratable si no se acota el problema) para encontrar la mejor secuencia de movimientos para ganar el juego.
- Robots móviles: Permiten a los robots móviles descubrir cuál es el mejor camino para alcanzar una cierta meta, por ejemplo, podría ser útil para salir de un laberinto o para navegar en una casa con múltiples habitaciones.

Uno de los problemas del aprendizaje por refuerzo es su intratabilidad en la gran mayoría de problemas debido al habitual problema de la recompensa retrasada. Es decir, en la mayoría de dominios el aprendiz conoce cuál es el beneficio de la acción que realizó en un determinado estado tras una (larga) serie de acciones; no conoce el beneficio de seguir una acción inmediatamente a su realización. Por ejemplo, en el juego del ajedrez no es posible conocer el beneficio de mover un peón en mitad de la partida, hasta que ésta termina y se sabe el resultado.

3.2.3 Aprendizaje supervisado y no supervisado

El aprendizaje inductivo puede ser clasificado en supervisado y no supervisado.

En aprendizaje inductivo supervisado el modelo acepta como datos de entrada ejemplos clasificados, es decir, se conoce cuál es la clase a la que pertenecen. Los algoritmos de aprendizaje inductivo supervisados, con ayuda de una serie de heurísticas o reglas, generarán diferentes hipótesis, generando finalmente como salida aquella que mejor describa a los ejemplos.

Un ejemplo de aprendizaje supervisado consiste en clasificar a posibles clientes de un banco en morosos y no morosos, utilizando un modelo construido con la base de datos de clientes (que ya se sabe si han sido morosos o no).

En contraposición, en aprendizaje inductivo no supervisado (también denominado agrupación o clustering) los datos suministrados contienen ejemplos que no están clasificados, por tanto, el objetivo de los algoritmos que realizan este tipo de aprendizaje consiste en autodescubrir la mejor manera de separar/aprender dichos ejemplos. En aprendizaje supervisado el aprendizaje era guiado por el contraste que se llevaba a cabo con los ejemplos, en este caso, el aprendizaje es guiado por la semejanza/diferencia entre los datos. El resultado del aprendizaje es un conjunto de clusters o particiones de los datos, cuyos ejemplos en cada una de esas particiones tienen ciertas propiedades en común.

Un ejemplo de aprendizaje no supervisado consiste en encontrar grupos (o clusters) de clientes con gustos similares entre los clientes de una tienda de libros.

3.2.4 Aprendizaje conexionista

En este paradigma de aprendizaje el sistema es una red de nodos conectados entre sí. Sus arcos están etiquetados con valores y se dispone de una regla de propagación de valores. Ante cada estímulo, el sistema reacciona modificando los pesos de los arcos. Por tanto, se dice que el sistema aprende cuando adapta sus pesos de tal manera que la salida del sistema proporciona un mayor beneficio (por el motivo que sea) al mismo.

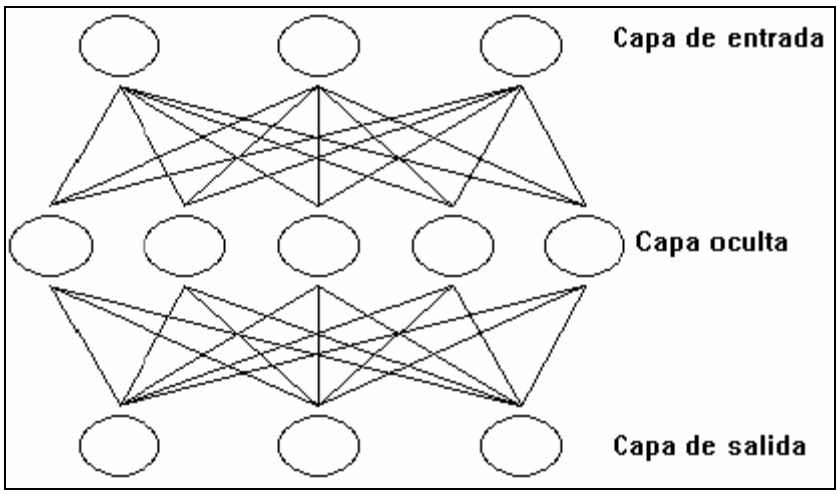

Ilustración 3.2 – Esquema de una red neuronal

3.2.5 Aprendizaje lógico inductivo

Cuando queremos aprender de conocimiento que no esté representado como atributo-valor o bien cuando el modelo o conjunto de reglas no permitan expresar el

conocimiento adquirido como consulta de valores de atributos. En estos casos es utiliza la programación lógica inductiva (*ILP*, Inductive Logic Programming), puede considerarse como la combinación del aprendizaje inductivo y la programación lógica.

La mejor forma de ilustrar la ventaja de la programación lógica inductiva sobre ciertos conjuntos de datos es a través de un ejemplo.

A0	A1	CLASE
0	0	+
0	1	-
0	2	-
1	0	-
1	1	+
1	2	-
2	0	-
2	1	-
2	2	+

Tabla 3.1 – Conjunto de datos que resuelve óptimamente *ILP*

Como puede observarse en la tabla 3.1, la clase es positiva cuando el valor de los atributos coincide, es decir, cuando se cumple la siguiente regla:

si A0=A1, entonces clase=+

Dicha regla es fácilmente expresable con lógica matemática pero no es expresable de manera resumida con conjuntos de reglas o pares de atributo-valor (los cuales no permiten comparar dos atributos), por tanto, la clasificación se resolvería con un conjunto de reglas más numeroso, por ejemplo:

si A0=0 y A1=0, entonces clase = +

si A0=1 y A1=1, entonces clase = +

si A0=2 y A1=2, entonces clase = +

en otro caso clase = -

3.3 Algoritmos de aprendizaje automático

En la actualidad, existen una gran cantidad de algoritmos de aprendizaje automático que, en función de los modelos que proporcionan, son clasificados en algoritmos de clasificación, agrupamiento (clustering) y asociación.

Los algoritmos de clasificación se enmarcan dentro del aprendizaje supervisado, mientras que los algoritmos de agrupamiento y asociación se enmarcan en el aprendizaje no supervisado.

3.3.1 Algoritmos de clasificación

Los algoritmos de clasificación proporcionan modelos que clasifican las nuevas instancias con un valor concreto para su clase, nominal o numérica. Pueden distinguirse varios tipos de algoritmos de clasificación en función de las características del modelo que proporcionan o de la fundamentación teórica en que se sustentan:

- **Algoritmos de reglas**: El modelo que proporcionan son una regla o conjunto de reglas ordenadas por importancia que clasifican una determinada instancia cuando sus atributos cumplen una regla.

  El algoritmo más conocido y simple es ZeroR cuya única regla consiste en clasificar a todas las nuevas instancias como perteneciente a la clase más frecuente.

  Otro conocido algoritmo es PART. Este algoritmo construye árboles C4.5 durante el proceso de aprendizaje convirtiendo, en cada paso, su mejor hoja en una regla.

```
plas <= 127 AND
mass <= 26.4 AND
preg <= 7: tested_negative (117.0/1.0)

plas > 154 AND
mass > 29.8: tested_positive (100.0/14.0)

…

plas <= 89 AND
plas > 0: tested_negative (13.0/1.0)

: tested_positive (252.0/105.0)
```

  Este ejemplo corresponde a la aplicación del algoritmo *PART* sobre el conjunto de datos *diabetes.arff*. Si los valores de los atributos de una nueva instancia cumplen la primera regla, esta instancia es clasificada como perteneciente a la clase negativa. Por el contrario, si ninguna de las reglas es cumplida por los valores de los atributos

de dicha instancia, esta es clasificada como perteneciente a la clase positiva (última regla).

- **Algoritmos de árboles**: Estos algoritmos generan un modelo en árbol (árbol de decisión o diagrama en árbol). Algunos de los más conocidos algoritmos son Id3, J48, M5P…

  Los árboles de decisión sin fáciles de entender (siempre que su tamaño no sea excesivo), usan un modelo de caja blanca (puedes entender fácilmente el por qué se ha tomado una determinada decisión) y, por último, pueden ser utilizados conjuntamente con otras técnicas de decisión.

  Id3 particiona el espacio de manera lineal teniendo como objetivo maximizar la ganancia de información (reducir la entropía) con cada nueva división.

  A modo de ejemplo se muestra el modelo (árbol) simplificado generado por J48 para el dominio *diabetes.arff*:

```
plas <= 127
|   mass <= 26.4: tested_negative (132.0/3.0)
|   mass > 26.4
|   |   age <= 28: tested_negative (180.0/22.0)
|   |   age > 28
|   |   |   plas <= 99: tested_negative (55.0/10.0)
|   |   |   plas > 99
|   |   |   |   pedi <= 0.561: tested_negative (84.0/34.0)
|   |   |   |   pedi > 0.561
|   |   |   |   |   preg <= 6
|   |   |   |   |   |   …
|   |   |   |   |   preg > 6: tested_positive (13.0)
plas > 127
|   mass <= 29.9
|   |   plas <= 145: tested_negative (41.0/6.0)
|   |   plas > 145
|   |   |   age <= 25: tested_negative (4.0)
|   |   |   age > 25
|   |   |   |   age <= 61
|   |   |   |   |   mass <= 27.1: tested_positive (12.0/1.0)
|   |   |   |   |   mass > 27.1
|   |   |   |   |   |   …
|   |   |   |   age > 61: tested_negative (4.0)
|   mass > 29.9
|   |   plas <= 157
|   |   |   pres <= 61: tested_positive (15.0/1.0)
|   |   |   pres > 61
|   |   |   |   age <= 30: tested_negative (40.0/13.0)
|   |   |   |   age > 30: tested_positive (60.0/17.0)
|   |   plas > 157: tested_positive (92.0/12.0)
```

- **Algoritmos de Bayes**: Utilizan el teorema de Bayes en su toma de decisiones para clasificar. Existen algoritmos que utilizan Naive Bayes, redes bayesianas, que realizan regresión simple bayesiana…

- **Algoritmos perezosos**: Este tipo de algoritmos no generan modelos de manera explícita como el resto, sino que retrasan la toma de decisiones hasta que reciben una instancia de test a clasificar.

  El algoritmo más conocido es *KNN* (K-Nearest Neighbour, K vecinos más cercanos en castellano), dicho algoritmo no construye ningún modelo, sino que simplemente clasifica cada instancia de test de acuerdo con la clase mayoritaria de sus K vecinos más cercanos, siendo K un parámetro configurable. La ilustración 3.4 muestra un ejemplo:

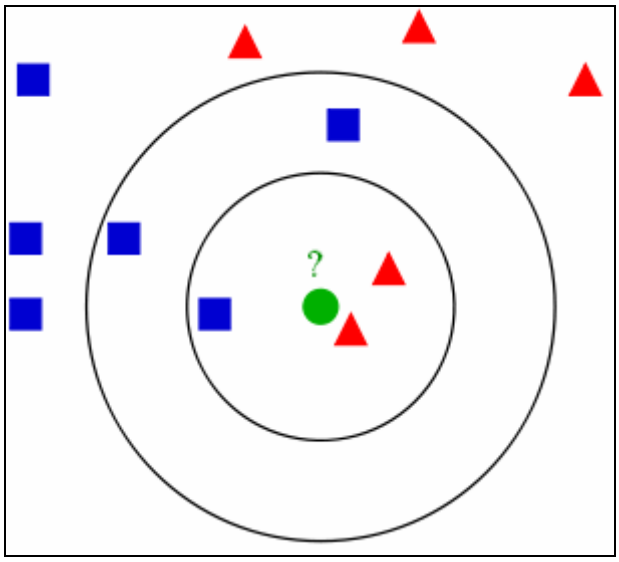

Ilustración 3.3 – Ejemplo de clasificación con *KNN*

  Supongamos que la instancia de color verde es una instancia de test cuya clase desconocemos. La instancia sería clasificada como perteneciente a la clase triángulo si el número de vecinos considerados fuera tres, y como perteneciente a la clase cuadrado si el número de vecinos considerados fuera de cinco, ya que tres de sus vecinos (mayoría) pertenecen a la clase cuadrado.

- **Meta-algoritmos**: Un meta-algoritmo es un algoritmo que utiliza uno o varios algoritmos base, cuyos resultados utiliza conjuntamente para dar una respuesta (clasificación o regresión) más precisa. El resultado suele ser más exacto (siempre que el algoritmo base prediga mejor que el azar).

3.3.2 Algoritmos de agrupamiento

Los algoritmos de agrupamiento o clustering realizan un aprendizaje no supervisado construyendo grupos o clusters en los cuales clasifican a determinadas instancias que tienen características comunes.

El algoritmo de agrupamiento más conocido es *K-means* (K-medias en castellano), su funcionamiento es muy sencillo:

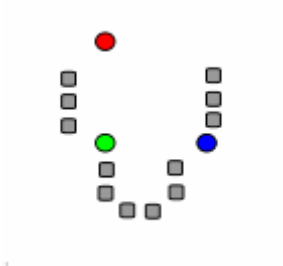

Ilustración 3.4 – Primer paso del algoritmo K-means

En primer lugar el usuario debe seleccionar el valor de K, es decir, el número de grupos o clusters en que desea dividir las instancias del conjunto de datos. Una vez elegido dicho número, el algoritmo selecciona aleatoriamente K instancias y pasa al segundo paso.

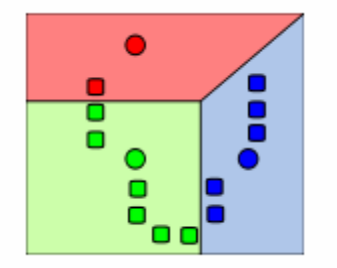

Ilustración 3.5 – Segundo paso del algoritmo K-means

En segundo lugar, las instancias seleccionadas en el paso anterior son consideradas los centroides de los clusters y el resto de instancias del conjunto de datos se clasifican como pertenecientes a la clase del centroide más cercano.

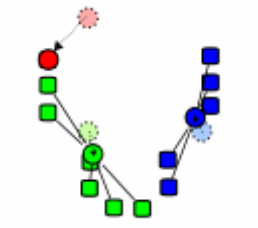

Ilustración 3.6 – Tercer paso del algoritmo K-means

Una vez creada la primera aproximación de lo que serán los clusters definitivos, el centroide es modificado siendo elegido como nuevo centroide la instancia que ocupa el lugar central.

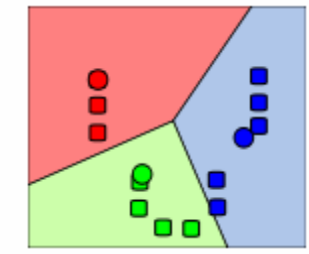

Ilustración 3.7 – Cuarto paso del algoritmo K-means

El cuarto paso no es un paso único, sino que consta de la repetición de los pasos segundo y tercero hasta que se alcance un equilibrio y no haya más modificaciones en la estructura y composición de los grupos.

3.3.3 Algoritmos de asociación

Los algoritmos que aprenden por asociación generan modelos tras el descubrimiento de relaciones entre los valores que toman las variables en grandes conjuntos de datos.

El algoritmo más conocido es *Apriori*; a continuación se muestra las reglas descubiertas cuando se aplica al conjunto de datos *weather.nominal.arff*:

1. outlook=overcast 4 ==> play=yes 4 conf:(1)

2. temperature=cool 4 ==> humidity=normal 4 conf:(1)

3. humidity=normal windy=FALSE 4 ==> play=yes 4 conf:(1)

4. outlook=sunny play=no 3 ==> humidity=high 3 conf:(1)

5. outlook=sunny humidity=high 3 ==> play=no 3 conf:(1)

6. outlook=rainy play=yes 3 ==> windy=FALSE 3 conf:(1)

7. outlook=rainy windy=FALSE 3 ==> play=yes 3 conf:(1)

8. temperature=cool play=yes 3 ==> humidity=normal 3 conf:(1)

9. outlook=sunny temperature=hot 2 ==> humidity=high 2 conf:(1)

10. temperature=hot play=no 2 ==> outlook=sunny 2 conf:(1)

Es decir, la regla 3 dice que cuando la humedad es normal y no hay viento se suele jugar al tenis, la regla 5 dice que cuando el tiempo es soleado y la humedad alta no se jugó al tenis…

3.4 Minería de datos

La minería de datos (*Data Mining*, en inglés) engloba un gran conjunto de técnicas y métodos que tienen como objetivo extraer información no trivial que se cumple en un conjunto de datos. Es decir, convertir los datos de entrada en conocimiento para permitir la toma de decisiones.

Las tareas que lleva a cabo la minería de datos con los conjuntos de datos son:

- **Predicción**: Como su propio nombre indica, trata de predecir el valor de la clase de una determinada instancia, dependiendo de si la clase es numérica o nominal se habla de:
  - **Clasificación**: Cuando la clase es nominal, el resultado de la predicción es uno de los posibles valores nominales que puede tomar la clase.
  - **Regresión**: Cuando la clase es numérica; el resultado es un valor numérico real.
- **Asociación**: Busca relaciones entre variables. Por ejemplo, si una persona compra leche, también compra huevos.
- **Agrupamiento (*Clustering*)**: El algoritmo intentará realizar grupos en los que instancias similares se incluyen en el mismo grupo.

La fundamentación teórica de la minería de datos se encuentra en la inteligencia artificial, principalmente el aprendizaje automático, las bases de datos y la estadística.

3.4.1 Proceso

El proceso de minería de datos que comienza con la adquisición de los datos en bruto y finaliza con la utilización del conocimiento adquirido es conocido como proceso *KDD* (*Knowledge Discovery in Databases*, Descubrimiento de Conocimiento en Bases de Datos, en castellano).

En la ilustración 3.9 se muestran las fases del proceso *KDD*:

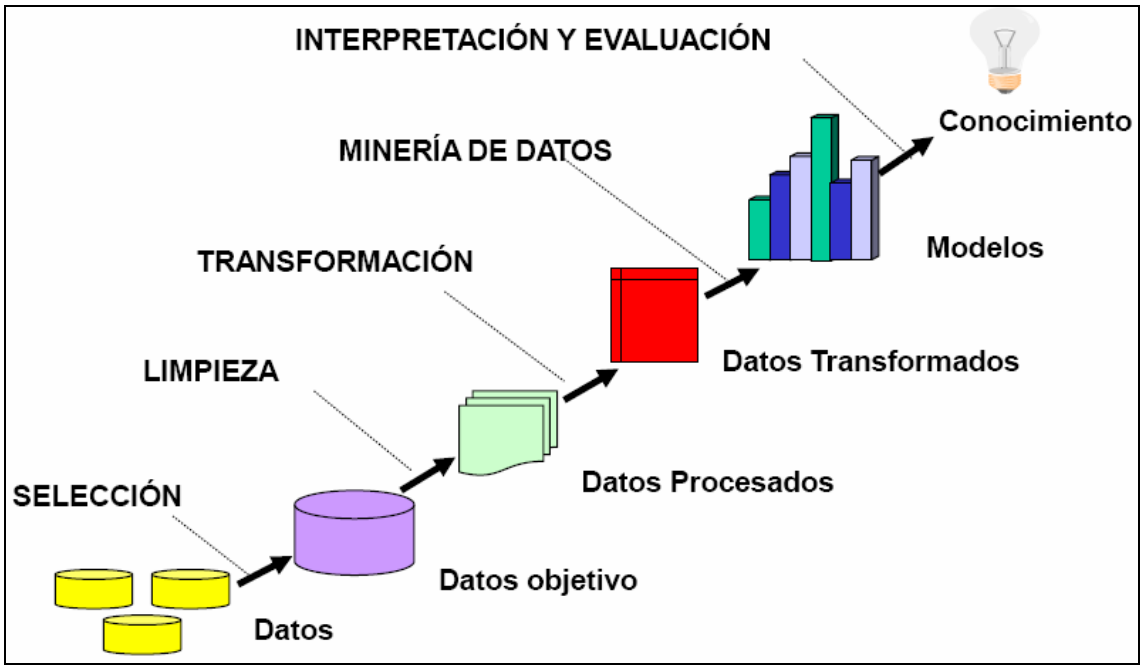

Ilustración 3.8 – Fases del proceso *KDD*

El proceso *KDD* es un proceso interactivo e iterativo (prueba y error) que consta de las siguientes fases:

- **Selección e integración del conjunto de datos**: Recopilación de los datos por diversas fuentes y almacenarlos en un formato común. Esta tarea puede suponer la mayor parte del tiempo. El objetivo es crear una única tabla de datos.
- **Limpieza**: Procesamiento de los datos que elimina aquellos erróneos y/o inconsistentes.
- **Transformación**: En ocasiones, es necesaria una transformación de los datos, tal como: convertir datos numéricos a categóricos o viceversa, reducción de la dimensionalidad, normalización de valores, simplificación de datos (por ejemplo, trabajar con metro en lugar de milímetros)…
- **Minería de datos**: En este fase se aplica una o varias técnicas de minería de datos, seleccionadas en función del modelo que se desea obtener. Una vez los datos han sido seleccionados, limpiados y transformados, la técnica de minería de datos empleada podrá aprender de los datos de una manera más eficiente. En la ilustración 3.10 se muestran las técnicas de minería de datos:

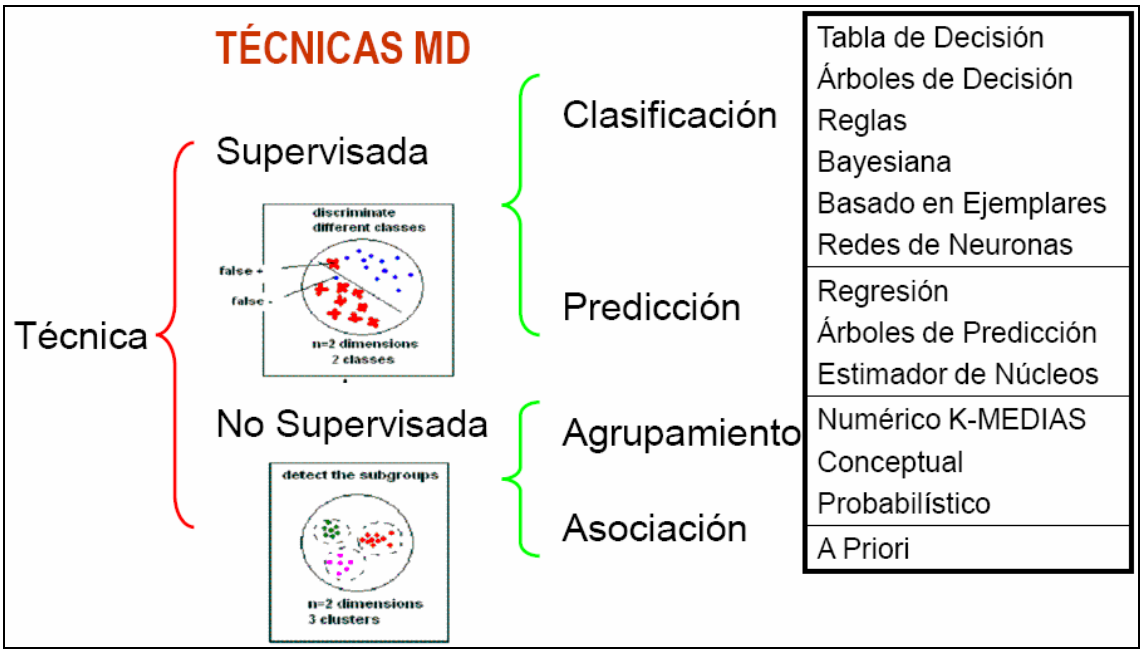

Ilustración 3.9 – Técnicas de minería de datos

- **Interpretación y evaluación**: En esta última fase se analiza el modelo/s obtenido/s para descubrir patrones válidos, nuevos, potencialmente útiles y comprensibles en el conjunto de datos. Además, los resultados deben evaluarse con un conjunto de test.

3.4.2 Herramientas

Cuando se trabaja con minería de datos disponer de un conjunto útil de herramientas puede simplificar bastante el trabajo. Las más importantes herramientas que pueden ser utilizadas son:

- **Herramientas de visualización**: Una inspección visual de los datos puede ser, en muchas ocasiones, la forma más rápida de obtención de conocimiento de los mismos. Además, este tipo de herramientas no sólo ayudan en la última fase, sino que disponer de ellas en la primera fase, permitirá un mejor conocimiento de los datos que deben seleccionarse, limpiarse y transformarse.

  Entre las herramientas más utilizadas se encuentran los histogramas y los diagramas de dispersión.

- **Data Warehouse**: Permite una base de datos común que simplifica el trabajo debido a que regula la consulta y modificación de los datos según los distintos perfiles, permite un mejor mantenimiento…

- Otras herramientas útiles son los **métodos OLAP**, los **sistemas OLTP** y **programas estadísticos**.

3.4.3 Aplicaciones

La minería de datos se utiliza en numerosos ámbitos, siendo algunos de ellos:

- Financieras y banca: Detección uso fraudulento de tarjetas de crédito…
- Análisis de mercado: Análisis de cesta de la compra, segmentación de clientes…
- Recursos humanos: Detección de las características de los empleados de mayor éxito.
- Medicina: diagnóstico de enfermedades
- Clasificación automática de páginas web para construir directorios
- Reconocimiento de caracteres escritos, de voz…
- Predicción de la demanda de electricidad, de gas...
- Detección de correo basura (spam)

3.5 Aplicaciones del aprendizaje automático

El aprendizaje automático está siendo aplicado en un gran abanico de áreas, algunas de las cuales son:

- Juegos
- Diagnósticos médicos
- Robótica
- Minería de textos
- Detección de fraudes en tarjetas de crédito
- Antivirus
- Análisis de la bolsa de valores
- Clasificación de imágenes
- Reconocimiento del habla
- Reconocimiento del lenguaje escrito
- Motores de búsqueda

En resumen, el aprendizaje automático puede ser utilizado cuando concurren alguna o varias de las siguientes características:

- La aplicación es difícil de desarrollar (reconocer complejos patrones: caras…)
- Aplicaciones autoadaptables, tales como asistentes personales, filtros anti-spam…
- Minería de datos, es decir, se desea llevar a cabo un análisis profundo de un conjunto datos.

4. Fundamentos de la Computación Evolutiva

En este capítulo se expondrán los fundamentos de la computación evolutiva, una de cuyas aplicaciones es el presente proyecto. En primer lugar se realizará una introducción sobre sus fundamentos e historia, posteriormente se explicarán los tres paradigmas en que se compone la misma (algoritmos genéticos, estrategias evolutivas y programación genética) y, por último, se expondrán las críticas que suscita/ha suscitado.

4.1 Introducción

4.1.1 Fundamentos

Hasta hace relativamente poco tiempo, la creencia universal con respecto al origen de las especies era que Dios las había creado una por una y permanecían inmutables desde entonces. Por si esto fuera poco, también se creía en la existencia de una jerarquía entre las especies, de tal modo que se creía que el ser humano ocupaba el más alto escalafón de acuerdo con la idea creacionista, habiendo sido creados el resto de seres vivos para su beneficio. A pesar de que el registro fósil indicaba la falsedad o, al menos, incompletitud de esta explicación, era tal el poder del cristianismo en Occidente que imposibilitó la aceptación del registro fósil.

En el siglo XVIII, el matemático, naturalista, biólogo y botánico francés, Georges Louis Leclerc[1], especuló con la posibilidad de que las especies se hubieran formada entre sí, destacando que hubo de haber un antepasado común entre el mono y el hombre. Sin embargo, no logró encontrar una respuesta lógica a sus teorías.

A principios del siglo XIX, el biólogo francés Jean-Baptiste Lamarck propuso una de las primeras teorías de la biología evolutiva. Según Lamarck, las especies descendían unas de otras, progresando gracias a la herencia que permitía la transmisión de padres a hijos de las características adquiridas en el medio por un organismo (hasta el momento de su reproducción).

A pesar de la fascinación del ser humano con la vida y su origen (prácticamente todas las religiones dan una explicación sobre la misma), no fue hasta la publicación en 1859 de *The Origin of Species* [2] (El Origen de las Especies) por parte del naturalista británico Charles Darwin que se tuvo una explicación seria sobre el origen de las especies y su evolución en el tiempo.

Charles Darwin[2] proponía que todas las especies descendían de un antepasado común y que iban evolucionando mediante selección natural. En la idea de selección natural

[1] Más conocido como Conde de Buffon (7 de Septiembre de 1707 – 16 de Abril de 1788)
[2] Shrewsbury, 12 de Febrero de 1809 – 19 de Abril de 1882, Kent

tuvieron influencia las ideas de Robert Malthus, que sostenía que el progreso humano es imposible sin la competición (por el alimento, por individuos del sexo opuesto…) y, puesto que los recursos son finitos, siempre ha habido competencia. Sin embargo, a pesar de la buena acogida de la evolución por parte de la comunidad científica de la época, no fue hasta la década de 1930 que su idea de selección natural fue considerada.

El científico alemán August Weismann, gran defensor de la selección natural, propuso que la información hereditaria se transmitía a través de un cierto tipo de células y, puesto que dicha teoría era incompatible con la teoría lamarckiana, llevó a cabo un experimento para desmentirla de manera científica. Para ello cortó la cola a 22 generaciones de ratas sin que las ratas acabaran por perder esa extremidad.

Entre 1857 y 1868 el monje austriaco Gregor Johan Mendel realizó experimentos con guisantes describiendo sus tres leyes de la herencia genética. Sin embargo, no fue hasta después de su muerte que su teoría

En 1896 el psicólogo norteamericano James Mark Baldwin planteó el conocido como efecto Baldwin, que afirma que, puesto que el aprendizaje beneficia la supervivencia, aquellos individuos con mayor capacidad de aprendizaje tendrán más descendencia, repercutiendo en la frecuencia de genes responsables del aprendizaje.

En 1953 James Watson y Francis Crick descubrieron la estructura en doble hélice del ADN. Posteriormente fue descubierto el ARN.

4.1.2 Historia

En 1932 el psicólogo norteamericano y profesor en el Harvard Medical School, Walter Bradford Cannon, publicó *The Wisdom of the Body* [*14*] (La Sabiduría del Cuerpo), en el que comparaba la evolución natural con un proceso de aprendizaje.

El celebérrimo matemático Alan Turing, en su artículo *Intelligent Machinery* [*15*] (Maquinaria Inteligente), publicado en 1948 y más recalcadamente en su artículo *Computing Machinery and Intelligence* [*17*] (Inteligencia y Maquinaria Computacional), publicado en 1950 y considerado un clásico en la Inteligencia Artificial, vislumbraba una clara conexión entre la evolución y el aprendizaje de las máquinas.

Entre 1957 y 1960, el ingeniero aeronáutico e informático, Alexander G. Fraser publicó tres artículos [*19*, *20*, *21*] sobre la evolución de sistemas biológicos por ordenador, acercándose bastante a lo que hoy conocemos por algoritmo genético. De hecho, en sus trabajos versa sobre una representación binaria, un cruce probabilístico…

R. M. Friedberg intentó [*23*] de manera poco exitosa, en 1958, hacer evolucionar programas de ordenador considerando más valiosas las instrucciones que llevaban a resultados más exitosos.

En 1954, el matemático Nils Aall Barricelli llevó a cabo las primeras simulaciones de sistemas evolutivos en computadores, fueron los comienzos de la vida artificial.

En 1956 G. J. Friedman [*25*] dio en su tesis los primeros pasos de la robótica evolutiva. Aplicó técnicas evolutivas a robótica empleando "retroalimentación selectiva" como selección natural. Sorprendentemente, los circuitos de control que empleaba eran muy similares a las modernas redes neuronales. Desarrolló una manera de construir y evaluar estos circuitos de control, de una manera similar a lo que hoy se denomina hardware evolutivo. Además [*24*], indicó que máquinas pensantes, como jugadores inteligentes de ajedrez podrían lograrse con reproducciones y mutaciones.

En 1958, el científico alemán Hans-Joachim Bremermann [*22*] descubrió la relevancia de la evolución en los procesos de optimización. Trabajo con el concepto de poblaciones y remarcó la utilidad de la coevolución.

Sin embargo, como se mencionó al comienzo del capítulo, la computación evolutiva dio lugar a tres paradigmas (algoritmos genéticos, estrategias evolutivas y programación genética) cuyo desarrollo será mencionado en sus respectivos apartados.

4.1.3 Fundamentos biológicos

En este subapartado se expondrán los fundamentos biológicos de la computación evolutiva, es decir, aquellos conceptos provenientes del campo de la biología que son comunes a los tres paradigmas de la computación evolutiva. Aquellos conceptos propios de cada paradigma serán mencionados en sus respectivos apartados.

En la vida real los seres vivos están compuestos por unos 60 elementos químicos, de ahí que estos elementos químicos sean denominados bioelementos o elementos biogénicos. Sin embargo, no todos los elementos químicos que componen un ser vivo determinan sus características. En concreto, sólo el material genético es necesario para representar un individuo. Es decir, la representación del ser humano es su secuencia genómica.

En nuestro caso, el material genético se encuentra mayoritariamente en el núcleo celular en forma de cromosomas. Los cromosomas son cadenas de ADN[3] formadas por bases nitrogenadas (adenina, timina, guanina o citosina). Se define como gen a una o varias bases nitrogenadas, secuenciadas o no, que repercuten en el individuo dotándole con

[3] Ácido desoxirribonucleico.

una o varias característica. Los genes constituyen, pues, las unidades funcionales de transmisión de características en el proceso evolutivo.

El ADN completo de un individuo es lo que se conoce como genotipo y su correspondencia física es el fenotipo. Por tanto, puede hablarse del fenotipo de un gen o de un ADN. Como ejemplo, el genotipo del color de los ojos correspondería con unas determinadas bases nitrogenadas en unas posiciones concretas de unos cromosomas determinados y su fenotipo sería el color de ojos del individuo poseedor de dicho ADN.

Se define el concepto de población como un conjunto de individuos de la misma especie, es decir, que pueden reproducirse entre ellos. Por tanto, tendríamos que un individuo es una solución del problema y una población sería un conjunto de soluciones, mejores o peores.

Existen dos formas de reproducción: asexual, en la que un solo individuo es capaz de procrear descendencia (idéntica, por lo general) y sexual, en la que son necesarios dos individuos de distinto sexo. Las ventajas de la reproducción sexual son obvias, produce individuos diferentes y provoca una mayor variedad genética en la población. La reproducción sexual será por tanto la empleada en la computación evolutiva.

No obstante, no todos los individuos de una población se reproducen, sino que sólo aquellos más aptos (más cercanos a la solución) tienen descendencia. Por este motivo, han de seleccionarse de nuestra población aquellos individuos que finalmente se reproducirán. Existen varias técnicas para llevar a cabo dicha selección:

- Proporcional al fitness: Los individuos tienen más probabilidades de ser seleccionados cuanto mayor sea su fitness (de una manera proporcional). De esta manera los individuos menos aptos también tienen posibilidad de reproducirse.
- Torneos: Se elige un tamaño de torneo, t. Se seleccionan t individuos aleatoriamente y se selecciona el mejor de ellos. Este individuo pasará a una población intermedia con los individuos seleccionados para reproducirse. Este proceso se repite tantas veces como individuos se desea que tenga dicha población intermedia.

  El tamaño de los torneos permite controlar la presión selectiva. Es decir, si el tamaño de los torneos es muy pequeño los individuos menos aptos tendrán más probabilidades de reproducción ya que dicho torneo será más sencillo de superar.

Una vez han sido seleccionados los individuos más apropiados para reproducirse, es necesario definir una técnica de cruce que permita, a partir de los progenitores, la generación de nuevos individuos. Además, debe implementarse una técnica de mutación que, al igual que ocurre en la Naturaleza, modifique ligeramente en algunos casos a los nuevos individuos de tal forma que se convierte en poseedor de nuevo material genético que no tenía por qué existir en los progenitores.

Cuando se generan nuevos individuos en una población, han de eliminarse tantos individuos como se han generado, de tal modo que el tamaño de la población permanezca constante. Se denomina política de reemplazo a aquella que determine qué individuos no seguirán en la población en la siguiente generación. Se distinguen, principalmente, dos tipos de políticas:

- Política de inclusión: Se unen los M individuos generados a los N individuos ya existentes y se eliminan los M individuos menos aptos. Por tanto, podría no tener oportunidades individuos recién generados.
- Política de inserción: Suponiendo que se han generado M individuos, se eliminan los M individuos menos aptos de la población de "progenitores" y se incluyen en la nueva población a los individuos recién generados. Con esta política todos los individuos generados habrán tenido, al menos, una oportunidad de reproducción.

La reproducción, cruce y mutación son denominados operadores evolutivos, ya que aceptan una población como entrada y su salida es una población modificada.

Una vez creada la población los individuos han de ser evaluados para determinar cuán buenos son en su medio natural con la competición con otros individuos. La función de fitness en la Naturaleza es bien conocida, sin embargo, cuando aplicamos la computación evolutiva en la resolución u optimización de problemas…

Se habla de que en una población hay elitismo cuando el mejor individuo de la misma siempre permanece en ella. Es decir, se da elitismo cuando la política de reemplazo que se siga no permita eliminar al mejor individuo para insertar a un nuevo individuo en ella. De esta manera, se evita que una solución subóptima encontrada no sea explotada (o utilizada en el resto del proceso de búsqueda).

Por último, las siguientes técnicas han sido utilizadas con éxito en la computación evolutiva:

- Coevolución: Se habla de coevolución cuando se dispone de dos (o más) poblaciones evolucionando en paralelo, de tal forma que la aptitud de un individuo está dada en términos de individuos de las restantes poblaciones. Por tanto, el criterio de aptitud se modifica con el tiempo, ajustándose dinámicamente a la población.
- Nichos: Cuando se utilizan nichos en computación evolutiva, se generan dos poblaciones que son evolucionadas de manera aislada y, cada cierto número de generaciones, se produce un intercambio de individuos entre ambas. Tiene la ventaja de que dicho intercambio puede llevar a la población un material genético nuevo que resulte beneficioso a la hora de utilizarse en la reproducción de un individuo de la población local.

4.2 Algoritmos Genéticos

En este apartado se explicará el primero de los grandes paradigmas en computación evolutiva. En primer lugar se describirá como surgieron y, posteriormente, su fundamentación teórica.

4.2.1 Historia

Los primeros pasos de los que hoy denominamos algoritmos genéticos fueron dados por el científico norteamericano y hoy conocido como padre de los algoritmos genéticos, John Henry Holland, a principios de la década de 1960 [*3*, *4*]. En dichos artículos *Concerning efficient adaptive systems* (Acerca de Sistemas Adaptativos Eficientes) y *Outline for a logical theory of adaptive systems* (Esquema para una Teoría Lógica de Sistemas Adaptativos), Holland investigaba con sistemas adaptativos, llamados así porque eran capaces de automodificarse como respuesta por la interacción con el entorno.

Holland tras investigar con sistemas biológicos simples, creía necesaria una competición e innovación constantes para sobrevivir en un mundo dinámico. Por tanto, sus sistemas adaptativos debían ser competidores e innovadores.

En la década de 1960, Holland estudió diversos tipos de cruce, mutación…, ideó la representación binaria y propuso la codificación Gray para resolver problemas de epístasis.

En 1975 publica un libro que marcó un hito en el campo de la computación evolutiva y más concretamente de los algoritmos genéticos, *Adaptation in Natural and Artificial Systems* [*5*] (Adaptación en Sistemas Naturales y Artificiales), en el que Holland expuso todas sus ideas sobre sistemas adaptativos. Propuso algunas ideas que hoy son útiles pero que en la época no pudieron ser probadas debido a la escasa potencia de cálculo de la época.

Desde 1975 a 1985, los algoritmos genéticos se popularizaron en el mundo académico y su estudio e interés fueron tales que en 1985 se organizó el primer Internacional Congress on Genetic Algorithms (Congreso Internacional sobre Algoritmos Genéticos) en Pittsburg (Pennsylvania).

A partir de este congreso, el uso de algoritmos genéticos en aplicaciones reales se generalizó, no sólo debido a una mayor aceptación y conocimientos por parte de los investigadores, sino a que el incremento de los recursos computaciones permitían su ejecución con una población suficientemente grande (a diferencia de la época en que Holland los investigaba).

4.2.2 Fundamentos teóricos

Las probabilidades de obtener solución para un problema que pretende resolverse con un algoritmo genético (y, en general, con cualquier paradigma de computación evolutiva) decrecen estrepitosamente si no se diseña una buena representación para los individuos que representen soluciones del problema. Además, una buena elección de operadores evolutivos (tipos de cruce, mutación…) y unas tasas de probabilidad adecuadas asociadas a los mismos son igualmente relevantes.

En un algoritmo genético los individuos son representados con cadenas fijas de caracteres, generalmente bits.

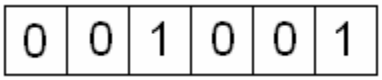

Ilustración 4.1 – Ejemplo de representación de un individuo

En algoritmos genéticos, debido a la representación de los individuos, se distinguen tres tipos de cruce:

- Cruce simple: Se elige una posición de la cadena y se genera un nuevo individuo que contiene, gen a gen, el material genético del padre hasta esa posición y el material genético de la madre a partir de dicha posición.

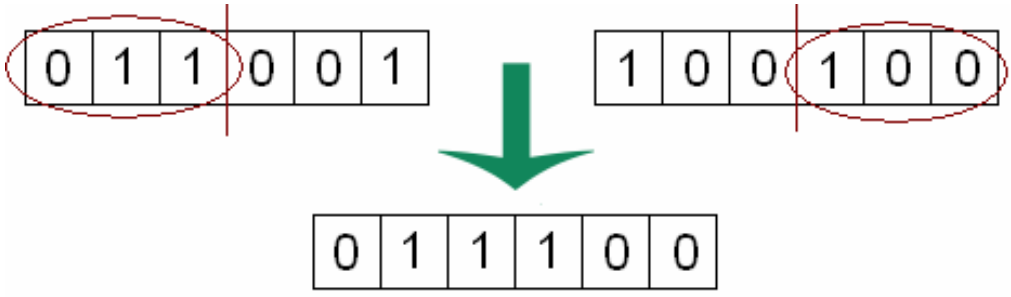

Ilustración 4.2 – Ejemplo de cruce simple

- Cruce multipunto: Equivalente a cruce simple, pero eligiendo más posiciones en la cadena, de tal forma que el material genético que tendrá el nuevo individuo es de la madre o del padre dependiendo entre cuáles de dichas posiciones se encuentre un determinado gen.

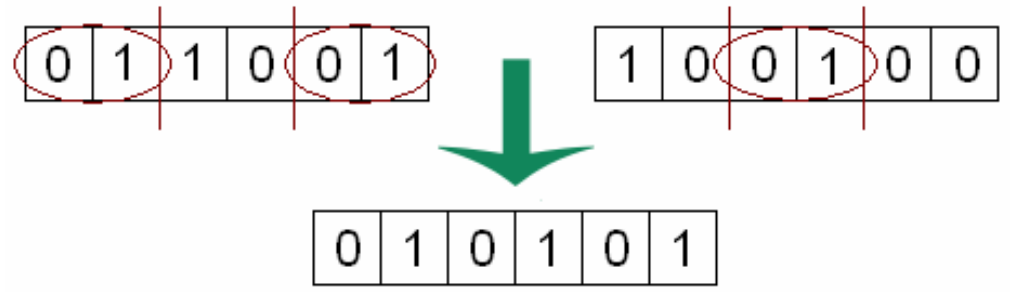

Ilustración 4.3 – Ejemplo de cruce multipunto (con dos puntos)

- Cruce uniforme: En este caso, el nuevo individuo contendrá genes del padre y de la madre de manera alternada.

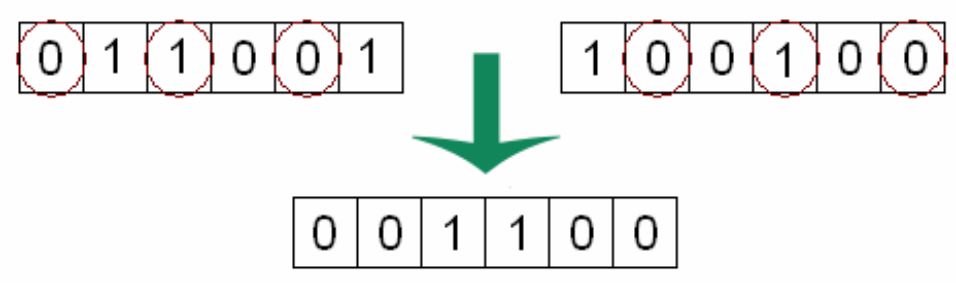

Ilustración 4.4 – Ejemplo de cruce uniforme

La mutación consiste en seleccionar un gen y asignarle un valor diferente (negar el valor si se trabaja con bits). Otra técnica de mutación es la inversión, que consiste en seleccionar un conjunto de genes consecutivos e invertir su orden en el cromosoma.

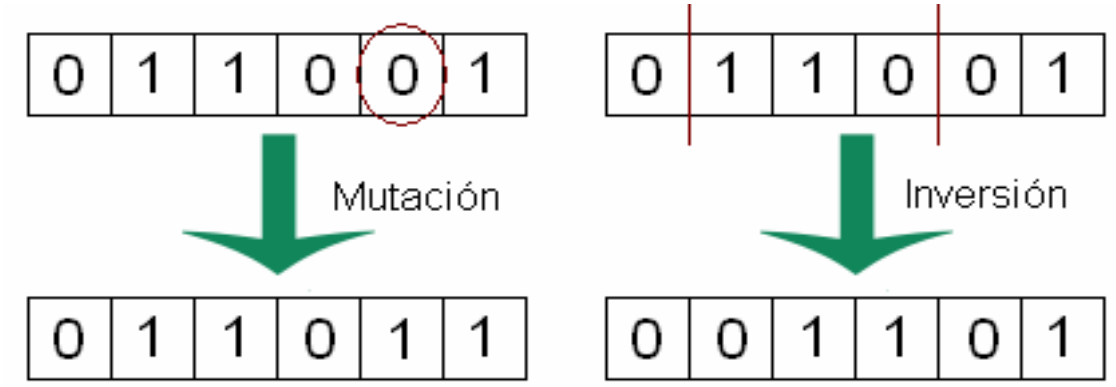

Ilustración 4.5 – Ejemplos de mutación e inversión

El esquema general de un algoritmo genético es el siguiente:

```
Inicio Algoritmo Genético
      Iniciar población de individuos (al azar)
      Evaluar población
      Durante un número de generaciones o hasta que se cumpla
      una condición
            Seleccionar individuos para reproducción
            Reproducir (y mutar)
            Evaluar nuevos individuos
            Insertar nuevos individuos en la población
      Fin Durante
Fin Algoritmo Genético
```

En primer lugar se inicializa una población de individuos. Cada individuo tiene un genotipo que se ha elegido de manera aleatoria y, generalmente equiprobable, entre el conjunto de genotipos posibles. En ciertas ocasiones, cuando se conoce de manera aproximada la posible solución/es, puede ser ventajoso una inicialización de la población con genotipos utilizando esta información. En segundo lugar, estos

individuos son evaluados para determinar cómo de bien resuelven el problema que se está tratando resolver.

Una vez que la población inicial ha sido creada, se entra en un bucle que finalizará cuando se cumpla una condición de parada (por ejemplo, haber encontrado la solución o no haberla mejorado en las últimas X generaciones) o al cabo de un número de generaciones.

En cada una de estas iteraciones se crea una población intermedia con los individuos seleccionados para reproducirse. Estos generan entre sí nuevos individuos (aplicando operadores de cruce y mutación) que han de ser evaluados para determinar, al igual que lo fueron sus progenitores, cuán bien están adaptados al medio.

Por último, antes de proseguir con una nueva evaluación del bucle, ha de actualizarse la población para la nueva generación siguiendo alguna política de reemplazo, ya que, de no seguirla, el tamaño de la población no dejaría de crecer y la búsqueda de la solución sería más complicada.

4.3 Estrategias Evolutivas

En este apartado se explicará el segundo de los grandes paradigmas en computación evolutiva. En primer lugar se describirá como surgió y, posteriormente, su fundamentación teórica.

4.3.1 Historia

En 1964, los investigadores Ingo Rechenberg, Paul Bienert y Hans Paul Schwefel desarrollaron las estrategias evolutivas en la Technische Universität Berlin (Universidad Técnica de Berlín). Estos investigadores estaban trabajando con problemas aerodinámicos complejos y necesitaban trabajar con valores reales.

La idea de utilizar un conjunto de valores reales asociados a un conjunto de varianzas para su mutación, resultó ser de gran éxito. En sus trabajos utilizaron una población de un solo elemento (1+1, en el siguiente apartado se describe esta técnica).

4.3.2 Fundamentos teóricos

Las estrategias evolutivas tienen una fundamentación teórica muy similar a la de los algoritmos genéticos. Cuando se habla de estrategias evolutivas, básicamente, se habla de algoritmos genéticos en los que se emplea una representación diferente. Además, emplear una representación diferente obliga a redefinir algunos conceptos, como el de cruce, mutación…

En estrategias evolutivas un individuo se representa con un número real (o un conjunto de ellos en los problemas cuya solución lo requiera), es decir, tiene una codificación continua. Esta codificación evita un problema que padecía la representación de los algoritmos genéticos y éste es que la codificación sólo podía tomar valores discretos y en un rango determinado.

En este tipo de codificación la mutación de los algoritmos genéticos no puede llevarse a cabo. La técnica de mutación se lleva a cabo con la distribución gaussiana (centrada en cero), funcionando del siguiente modo:

- El individuo necesitará codificar un valor para la varianza asociado a cada número real.
- Con cada varianza se obtiene un valor haciendo uso de la distribución gaussiana. Por tanto, en término medio, los valores generados serán mayores cuanto mayor sea el valor de la varianza.

- El valor obtenido en el paso anterior (es positivo o negativo de una manera equiprobable) es sumado a cada uno de los valores reales que codifica el individuo.

El valor de la varianza del individuo deberá ir decreciendo conforme la solución encontrada vaya acercándose a la óptima. De esta manera, los cambios en el valor que codifique el individuo serán menos bruscos y se "explotará" la solución, frente a una búsqueda más ciega que se produce cuando el valor de la varianza es elevado.

Como puede observarse, a diferencia de los algoritmos genéticos cuya fortaleza reside en el sobrecruzamiento, en las estrategias evolutivas el verdadero motor es la mutación. Por tanto, a diferencia de los algoritmos genéticos, con estrategias evolutivas es posible utilizar una población de un solo individuo.

La búsqueda de la solución de un problema con una población de un solo individuo (1+1) es un método de escalada puro y se procede de la siguiente manera:

- El individuo genera un nuevo individuo (se reproduce) por mutación, es decir, incrementa su parte funcional de acuerdo con el valor generado por la distribución normal con la varianza codificada. Su varianza también es mutada suponiendo que cuando se encuentran mejores soluciones en las últimas generaciones es debido a que se está lejos de la solución (de ahí que sea fácil mejorar), en contraposición a no encontrar mejores soluciones en las últimas generaciones, que se debería a que la solución encontrada es (casi) óptima y difícilmente mejorable.

  Por tanto, puesto que se desea que la varianza vaya decreciendo conforme nos acercamos a la solución, la varianza será multiplicada por una constante de decremento (c_d<1, cuyo valor estándar es 0.82) cuando se hayan encontrado pocas soluciones mejores en las últimas generaciones (parámetros configurables) y por una constante de incremento (c_i>1, cuyo valor estándar es 1.18) cuando se hayan encontrado muchas soluciones mejores en las últimas generaciones.
- Se evalúan los individuos con que cuenta la población en este paso intermedio.
- Se descarta al peor.

A modo de resumen, el esquema que se sigue es el siguiente:

```
Generación y evaluación de un individuo.
Durante un número de iteraciones o hasta cumplir una condición de parada
      Generar y evaluar un nuevo individuo.
      Descartar al menos apto.
      Mutar varianzas.
Fin Durante
```

No obstante, lo más habitual es utilizar estrategias evolutivas múltiples, es decir, con poblaciones de más de un elemento, ya que una población de un individuo es muy sensible a quedar "atrapada" en máximos locales.

En el caso de utilizar estrategias evolutivas múltiples tiene sentido definir un operador de cruce y deja de tener sentido la mutación de varianzas (es poco realista implementar la técnica empleada en 1+1). En esta técnica, el cruce entre dos individuos se define de la siguiente manera:

- El valor funcional del nuevo individuo se define como la media de los valores funcionales de los progenitores:

$$\vec{v}_{hijo} = \frac{1}{2}\left(\vec{v}_{padre} + \vec{v}_{madre}\right)$$

Ecuación 4.1 – Cálculo del valor funcional de los descendientes en EE[4]

- La varianza del nuevo individuo se calcula de la siguiente manera:

$$\vec{\sigma}_{hijo} = \sqrt{\left(\vec{\sigma}^2_{padre} + \vec{\sigma}^2_{madre}\right)}$$

Ecuación 4.2 – Cálculo del valor de la varianza de los descendientes en EE

La mutación de la parte funcional con estrategias evolutivas múltiples es idéntica, sin embargo, como se mencionó anteriormente, la mutación de las varianzas es diferente y se muta siguiendo la siguiente fórmula, con la que se consigue un decremento "paulatino":

$$\vec{v}_2 = \vec{v}_1 + N_0\left(\sigma_1\right)$$

Ecuación 4.3 – Cálculo del valor funcional en la mutación

El método de selección puede ser cualquiera de los disponibles con algoritmos genéticos: ruleta, ranking… Además, debe seguirse una política de reemplazamiento.

[4] Estrategias Evolutivas

4.4 Programación Genética

En este apartado se explicará el último de los grandes paradigmas que surgió en computación evolutiva. En primer lugar se describirá su historia y, posteriormente, su fundamentación teórica.

4.4.1 Historia

En 1985, el norteamericano Nichael Lynn Cramer propuso el uso de una representación arbórea para los individuos en computación evolutiva [6]. Sin embargo, el trabajo de Cramer no tuvo buena acogida ya que requería que la función de aptitud fuera determinada por el usuario de manera interactiva. En 1987 el científico alemán Jürgen Schmidhuber propuso desde Europa la misma idea de manera independiente.

John R. Koza [7] avanzó de manera decisiva en esta dirección (automatizó el cálculo de los fitness/aptitudes de los individuos). Utilizó programación genética en la resolución de complejos problemas de búsqueda y optimización.

A finales de la década de 1990, la mejora en las técnicas de programación genética y el aumento de recursos computacionales propiciaron su utilización en problemas de mucha mayor complejidad, tales como diseño electrónico, computación cuántica…

La Programación Genética es tan poderosa que no sólo ha encontrado soluciones muy similares a las que los humanos utilizan o utilizaron, sino que John R. Koza ha patentado más de 15 soluciones que este paradigma ha proporcionado para diferentes problemas.

Desde 1990 hasta principios del siglo XXI, la progamación genética ha evolucionado muy rápidamente, se han diseñado nuevas técnicas, ha sido exhaustivamente estudiada y se ha consolidado como un paradigma independiente en la computación evolutiva.

4.4.2 Fundamentos teóricos

La principal diferencia de este paradigma de la computación evolutiva con las restantes es la representación que hace de los individuos. La programación genética representa a los individuos con una estructura arbórea en que los nodos internos contienen funciones/operadores y los nodos hoja contienen símbolos terminales. Los operadores y terminales disponibles son definidos por el usuario (el usuario debe indicar también la aridad de cada función/operador).

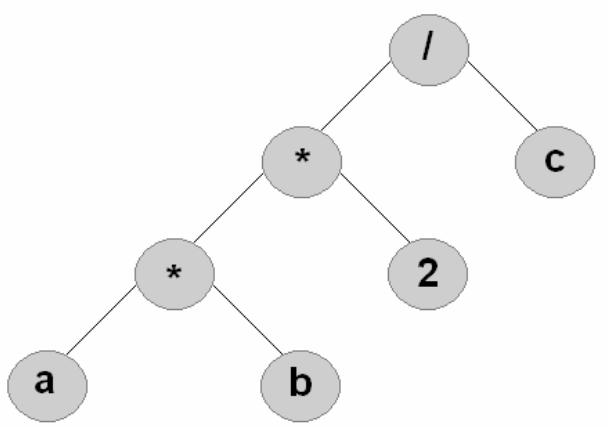

Ilustración 4.6 – Ejemplo de representación de un individuo

La representación arbórea es mucho más potente que la representación de los restantes paradigmas. Esta representación supone un salto cualitativo en cuanto a la potencia de representación de los individuos se refiere. En una estructura arbórea pueden codificarse soluciones con mayor semántica, por lo general: funciones matemáticas y programas de ordenador (en LISP o cualquier otro lenguaje).

A la hora de iniciar la población de individuos, han de definirse no sólo los conjuntos de funciones y terminales disponibles, sino también sus probabilidades de selección/aparición. Puede llevarse a cabo de dos maneras:

- Inicialización completa: El individuo se va generando añadiéndole todos sus nodos con funciones hasta que el nodo se encuentra a una determinada profundidad, en cuyo caso, se crea un nodo con un símbolo terminal.

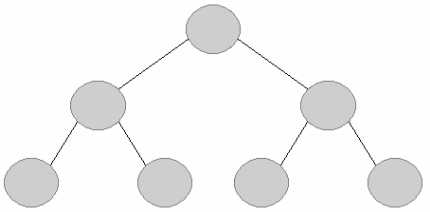

Ilustración 4.7 – Ejemplo de inicialización completa

- Inicialización creciente: El individuo se genera añadiendo cada vez un nodo (que puede ser una función o un terminal) hasta que, al menos, un nodo alcance la profundidad deseada.

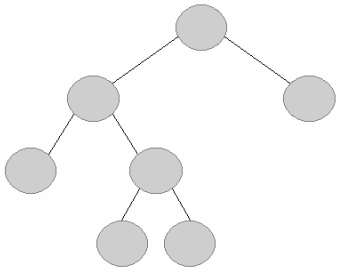

Ilustración 4.8 – Ejemplo de inicialización creciente

Otra posibilidad es crear la población en la que un determinado porcentaje (50% o menor) de los individuos se generen con una inicialización completa y los restantes con una inicialización creciente.

Puesto que la representación de los individuos es diferente, han de idearse nuevas técnicas para llevar a cabo la generación de nuevos individuos por reproducción y la mutación de los mismos.

Para la generación de nuevos individuos se selecciona un nodo en el árbol de cada progenitor y se generan dos individuos: el primero es idéntico al padre salvo el subárbol a partir del nodo seleccionado, que será idéntico al subárbol que tiene la madre a partir de su nodo seleccionado y para el segundo individuo generado el proceso es equivalente intercambiando los progenitores.

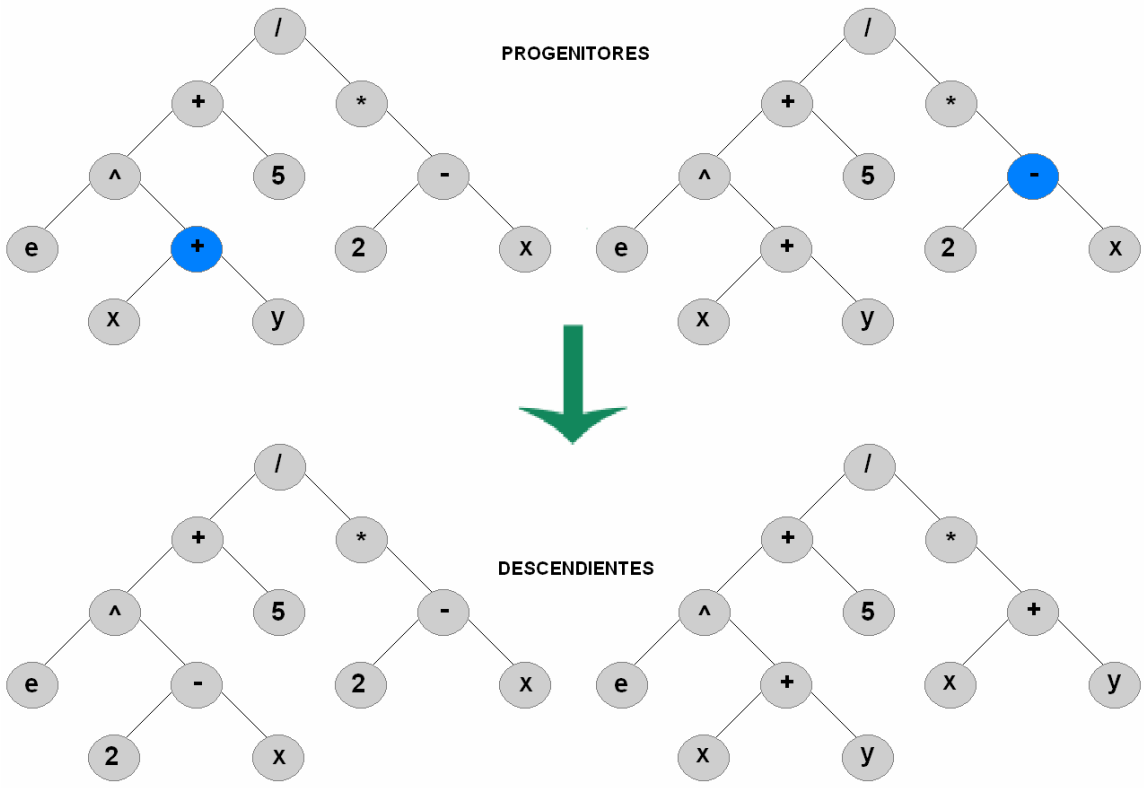

Ilustración 4.9 – Cruce de dos árboles idénticos

El punto de cruce es el nodo seleccionado y han de decidirse la probabilidad con que cada tipo de nodo puede ser seleccionado. Por ejemplo, con un 0.9 de probabilidad se selecciona un nodo interno y con un 0.1 para nodos terminales u hojas. Además, como puede observarse, a diferencia de los algoritmos genéticos, dos padres idénticos pueden generar dos individuos diferentes.

La mutación consigue obtener un nuevo individuo (programa o función matemática) a partir de un único progenitor. Se distinguen tres tipos de mutaciones:

- **Mutación funcional simple**: Se selecciona al azar una función en el árbol y se reemplaza por otra del conjunto de funciones disponibles.

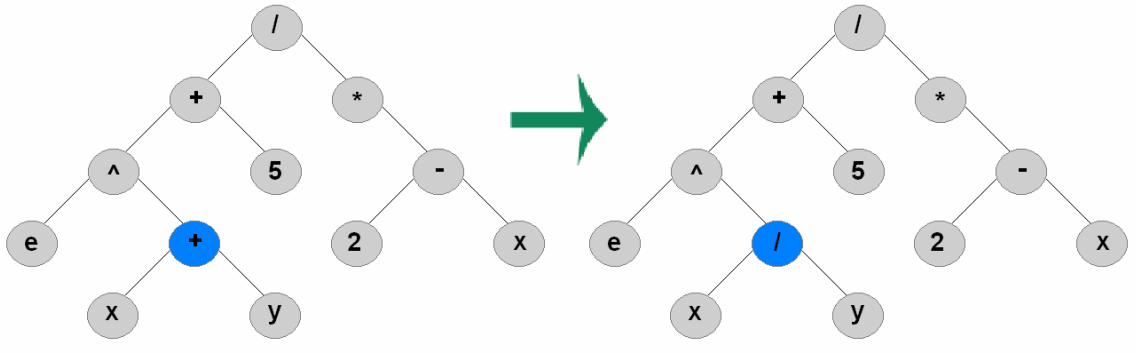

Ilustración 4.10 – Ejemplo de mutación funcional simple

- **Mutación terminal simple**: Equivalente a la mutación funcional simple, pero reemplazando una símbolo terminal por otro del conjunto de terminales especificado por el usuario.
- **Mutación de árbol**: Se selecciona un subárbol y se reemplaza por uno construido aleatoriamente (de la misma manera que se generan al iniciar la población).

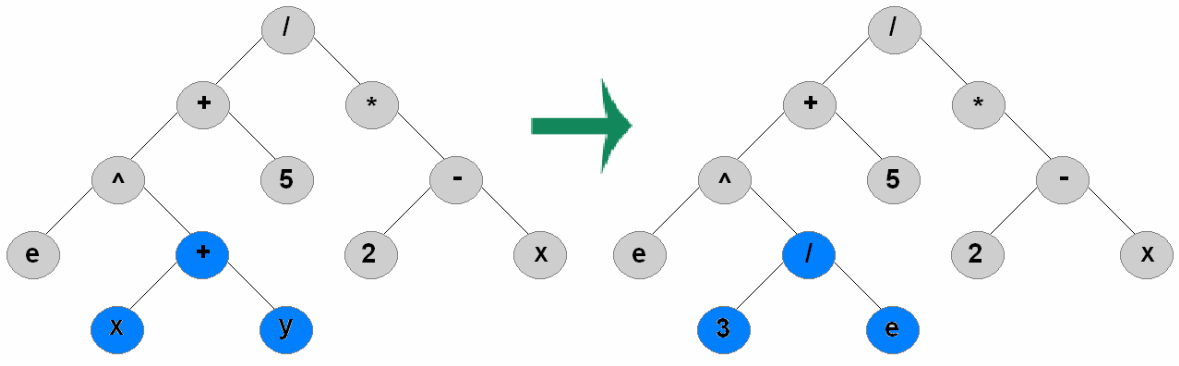

Ilustración 4.11 – Ejemplo de mutación de árbol

El operador de permutación definido en programación genética es similar al operador de inversión definido en algoritmos genéticos. Una permutación consiste en intercambiar dos nodos del mismo tipo (función o terminal) en el árbol.

Existe una mejora denominada encapsulación por la que se da nombre a un subárbol y puede utilizarse en cualquier nodo de ese árbol u otro como cualquier otra función o terminal. Este subárbol, denominado subrutina o ADF[5], puede acelerar la búsqueda de la solución en cierto tipo de problemas. Por ejemplo, podrían encontrarse subrutinas para calcular el área de una figura geométrica involucrada en el problema.

[5] Automatically Defined Function (Función Automáticamente Definida)

4.4.3 Problemática de la Programación Genética

Uno de los problemas de que adolece la programación genética es la dificultad que surge al trabajar con constantes numéricas. Esto es debido a que las constantes numéricas son símbolos terminales que deberían estar definidas por el usuario en el conjunto de símbolos terminales disponibles. Sin embargo, el usuario desconoce la solución y, por tanto, el valor que asignar a dichos terminales.

Una primera solución al problema sería definir un símbolo terminal que, lejos de ser una constante fija, generara una constante en un rango determinado de manera aleatoria. De este modo, la búsqueda de la solución seguiría siendo infructuosa en la mayoría de las ocasiones.

La técnica más empleada para solucionar este problema es construir un algoritmo híbrido, de tal modo que, simultáneamente, el árbol genere con programación genética y las constantes haciendo uso de algoritmos genéticos o estrategias evolutivas.

El segundo gran problema en el uso de la programación genética es el excesivo tamaño que alcanzar los árboles en relativamente pocas generaciones sin que su fitness aumente considerablemente (*bloat*). Este crecimiento puede llegar a ser exponencial y está causado, principalmente, por intrones. Un intrón es un subárbol que no hace nada.

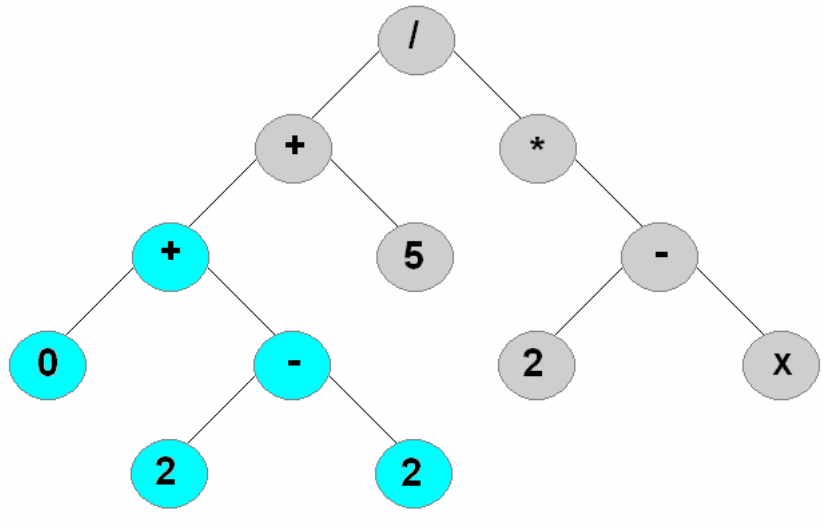

Ilustración 4.12 – Ejemplo de intrón

Este crecimiento excesivo causa que los individuos ocupen más tamaño en memoria, tarden más tiempo en ser ejecutados, sean menos legibles y que la búsqueda se estanque.

Para evitar el bloat se puede:

- Diseño de los operadores de mutación, cruce… para que no construyan individuos mayores.

- Penalización del fitness de los individuos de manera proporcional a su tamaño, de tal manera que un individuo muy grande sólo tendrá oportunidades de selección si su fitness es significativamente mayor.

$$F'(x) = F(x) + k \cdot nodos(x)$$

Ecuación 4.4 – Cálculo del fitness con penalización parsimoniosa

4.5 Problemática de la computación evolutiva

La computación evolutiva ha demostrado su valía en sus más de 30 años de historia principalmente en problemas de optimización. Además, no es necesario tener un conocimiento profundo del problema a resolver.

La computación evolutiva es muy útil en la resolución de problemas NP-Completos en los que nos conformamos no con encontrar la mejor solución, sino una buena solución en un tiempo determinado. Además, son especialmente valiosas en problemas de:

- Optimización (numérica, de consultas a bases de datos…)
- Planificación (de rutas, movimientos de robots…)
- Predicción (de demanda eléctrica…)
- Aprendizaje automático (reconocimiento de patrones…)

El uso de algoritmos evolutivos en la resolución de problemas no está exento de dificultades. En primer lugar, es necesaria la especificación de una gran cantidad de parámetros cuyo incorrecto “ajuste” puede anular las probabilidades de éxito.

Los tres paradigmas de la computación evolutiva pueden verse menoscabados dependiendo de la representación que se decida emplear: puede no ser lo suficientemente precisa (y/o completa, en el caso de la programación genética). Una buena codificación debe, aunque no siempre puede, ser:

- Completa: Todas las posibles soluciones deben tener cabida en ella.
- Coherente: No debe poder codificar soluciones imposibles.
- Colindante: Pequeños cambios en el genotipo han de correspondiente con pequeños cambios en el fenotipo.

El cálculo de la aptitud o fitness de los individuos no siempre es sencillo. Lo ideal no sólo sería que individuos diferentes tuvieran evaluaciones diferentes, sino que el fitness fuera aumentando paulatinamente conforme el individuo se fuera acercando en la dirección correcta a la solución, de tal manera que sea una búsqueda lo más dirigida posible. En la ilustración 4.13 se comparan la función de fitness ideal con una función de fitness abrupta:

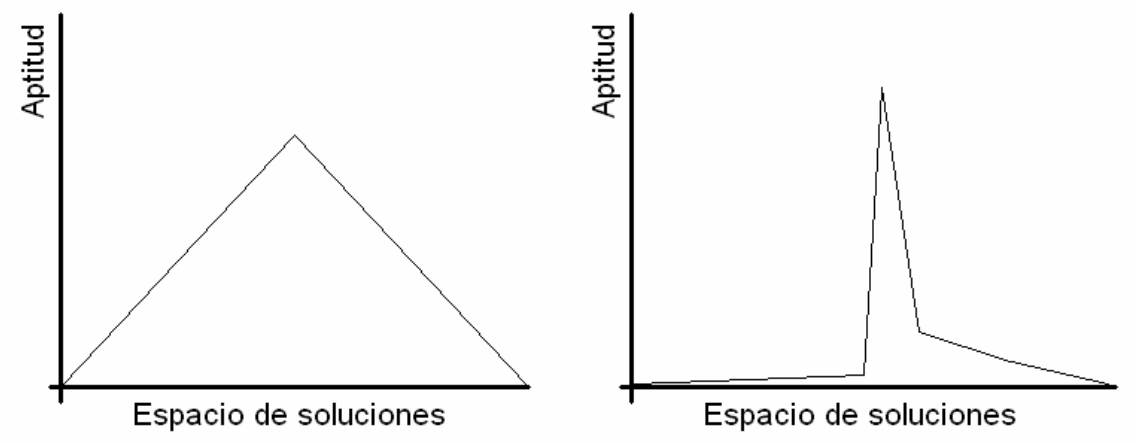

Ilustración 4.13 – Funciones de fitness ideal y abrupta

Otro problema importante que puede darse es la pérdida de diversidad, que se presenta cuando los individuos de la población muy parecidos entre sí (sin haber logrado la solución óptima), lo que provoca deriva genética y la búsqueda queda estancada en un máximo local.

5. Weka

En este apartado se hará una introducción al software libre *Weka* y se detallará el formato de los ficheros de entrada que acepta. Posteriormente, se detallará su funcionalidad para cada una de sus aplicaciones: explorador, experimentador, flujo de conocimiento y consola. Por último, se detalla brevemente la estructura de *Weka*, indicando cada uno de sus paquetes y describiendo sus más importantes clases. Es importante conocer la estructura interna de *Weka* porque una parte fundamental de este proyecto es la integración del algoritmo desarrollado dentro de *Weka*, para que pueda ser utilizado como un algoritmo más. Este capítulo se terminará con una explicación acerca de cómo se pueden integrar nuevos algoritmos en *Weka*.

5.1 Introducción

Weka, siglas de *Waikato Environment for Knowledge Analysis* (Entorno Waikato para el análisis de conocimiento, en castellano), es un software que contiene una colección de algoritmos de aprendizaje automático para tareas de minería de datos. Ha sido desarrollado en la Universidad de Waikato (Nueva Zelanda).

Ilustración 5.1 – Logo de la Universidad de Waikato

La aplicación está escrita íntegramente en Java y puede ejecutarse en UNIX, Windows y Macintosh. Se distribuye bajo licencia GPL[1], lo que significa que puede distribuirse libremente, pero cualquier modificación debe distribuirse obligatoriamente bajo esta misma licencia, por tanto, el trabajo realizado en el presente proyecto se distribuirá bajo licencia GPL.

El software debe su nombre al *Weka* (Gallirallus australis), un pájaro incapaz de volar, del tamaño de un pollo y muy indagador que solamente se encuentra presente en Nueva Zelanda.

[1] GNU General Public License. http://www.gnu.org/copyleft/gpl.html

Ilustración 5.2 – Logo de *Weka*

Desde la página web del proyecto: http://www.cs.waikato.ac.nz/ml/weka/, puede descargarse el software y consultar información adicional relacionada (desarrolladores, publicaciones…).

Su ejecución es muy sencilla, basta con descomprimirlo y ejecutar el archivo weka.jar (haciendo doble clic o con el comando *java –jar weka.jar*).

Una de las principales dificultades con que se encuentran los usuarios de *Weka* es la poca memoria que reserva, tan solo 100 MB. Para reservar más memoria se puede utilizar el comando:

java –Xms<mínima-memoria-asignada>M -Xmx<máxima-memoria-asignada>M -jar weka.jar

Donde –Xms y –Xmx indican, respectivamente, las cantidades mínima y máxima, expresadas en magabytes, de memoria RAM que se desean asignar.

Existen tres versiones de *Weka*:

- Book version (versión del libro, en castellano)
- Stable version (versión estable, en castellano)
- Developer version (versión de desarrollador, en castellano)

Cualquiera de las tres puede ser descargada para Windows, Mac OS X u otras plataformas (Linux, etc.).

Las diferencias entre las diferentes versiones son las siguientes:

- Versión del libro: Está vinculada al libro “Data Mining: Practical Machine Learning Tools and Techniques” escrito por los profesores del Departamento de Informática de la Universidad de Waikato Ian H. Witten y Eibe Frank. No ha sufrido cambios desde la publicación del libro en 2005 y sólo es actualizada para corregir bugs, pero no para añadirle nuevos algoritmos, filtros…
- Versión estable: Surge en Diciembre del año 2008 por escisión de la versión de desarrollador y, al igual que la versión del libro, sólo será actualizada para corregir bugs.
- Versión de desarrollador: Ideada para ser la utilizada por quienes quieran añadir nuevas funcionalidades. Es actualizada para corregirle sus bugs y, a diferencia de las anteriores, para añadirle nuevas funcionalidades (clasificadores, filtros…) De ahí que cualquier contribución al proyecto deba ser compatible con esta versión.

Este proyecto se desarrolla sobre la versión de desarrollador 3.5.8 en Windows.

5.2 Ficheros de datos

Los ficheros de datos que reconoce *Weka* y, por tanto, pueden ser cargados para ser utilizados son:

- ARFF[2] (*.arff)
- C4.5 (*.data o *.names)
- CSV (*.csv)
- libsvm (*.libsvm)
- Binary serialized instances (instancias serializadas en binario) (*.bsi)
- XRFF (*.xrff)

Los ficheros de datos con extensión arff son los ficheros más comunes y serán los que se usarán en el desarrollo del proyecto. Los comentarios de línea son iniciados por '%', los valores desconocidos se indican con '?' y en el fichero se ha de indicar:

- El nombre de la relación: *@relation <nombre-de-la-relación>*
- Declaraciones de atributos: *@attribute <nombre-del-atributo> <tipo>*. Donde el tipo de los atributos puede ser el siguiente:
  - Numérico: integer o real
  - Especificación nominal (valores especificados entre llaves)
  - Fecha: date <formato-fecha>[3]
  - String
- Sección de datos: *@data <datos>*. Donde se separan con espacios los valores de los atributos y con un salto de línea cada instancia.

A continuación se muestra, a modo de ejemplo, el contenido de un fichero de datos con extensión arff que contiene ocho instancias:

```
@relation pima_diabetes
%El nombre de la relación es pima_diabetes
@attribute 'preg' real
@attribute 'plas' real
@attribute 'pres' real
@attribute 'skin' real
@attribute 'insu' real
@attribute 'mass' real
@attribute 'pedi' real
@attribute 'age' real
```

[2] ARFF son las siglas de Attribute-Relation File Format (Formato de Fichero de Relación de Atributos, en castellano)

[3] La cadena por defecto acepta el formato ISO-8601 que combina fecha y hora de la siguiente manera: "yyyy-MM-dd'T'HH:mm:ss".

```
@attribute 'class' {tested_negative, tested_positive}
%Hay ocho atributos más el atributo clase
@data
%A continuación vienen las ocho instancias
6,148,?,35,0,33.6,?,50,tested_positive
1,85,66,29,0,26.6,0.351,31,tested_negative
8,183,64,0,0,23.3,0.672,32,tested_positive
1,89,?,?,?,?,0.167,21,tested_negative
0,137,40,35,168,43.1,2.288,33,tested_positive
5,116,74,0,0,25.6,0.201,30,tested_negative
3,78,50,32,88,31,0.248,26,tested_positive
10,115,0,0,0,35.3,0.134,29,tested_negative
```

5.3 Funcionalidad detallada

Una de las razones por las que *Weka* se ha convertido en una aplicación mundialmente conocida y utilizada es debido a que cubre el proceso *KDD* (Knowledge Discovery in Databases, Descubrimiento de Conocimiento en Bases de Datos en castellano) completo. Otra de las razones es que cuenta con una interfaz gráfica muy intuitiva. Cuando se ejecuta el programa nos encontramos con la ventana inicial.

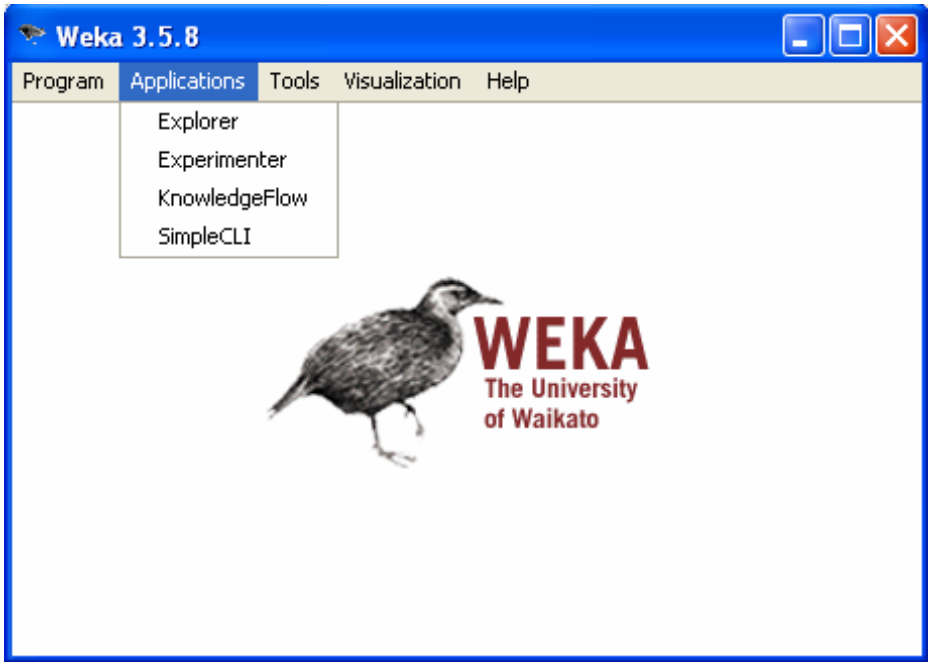

Ilustración 5.3 – Ventana inicial de *Weka*

La pestaña más importante es *Applications*, las restantes ofrecen funcionalidades adicionales, tales como: ayuda, uso de memoria, visor de *arff*[4]… Desde la pestaña *Applications*, puede elegirse una de cuatro posibles interfaces: Explorer, Experimenter, KnowledgeFlow y SimpleCLI. Cada una de ellas es descrita detalladamente en los siguientes apartados.

5.3.1 Explorer

Es la opción más utilizada y permite hacer operaciones de minería de datos sobre un archivo de datos con extensión *arff*. Puesto que la minería de datos es una ciencia experimental y no hay un clasificador universal ideal, en esta interfaz están disponibles un gran número de clasificadores y herramientas para preprocesar los datos. El explorador permite tareas de:

[4] Los conjuntos de datos que se le proporcionan a Weka tienen extensión arff. El visor de arff muestra el conjunto de datos en formato tabular.

- **Preprocesado de los datos**: Corresponde con la pestaña *Preprocess*. Ofrece diferentes maneras para cargar las instancias (desde un archivo, proporcionando una URL y desde una base de datos) y un conjunto de algoritmos para generar conjuntos de instancias aleatoriamente.

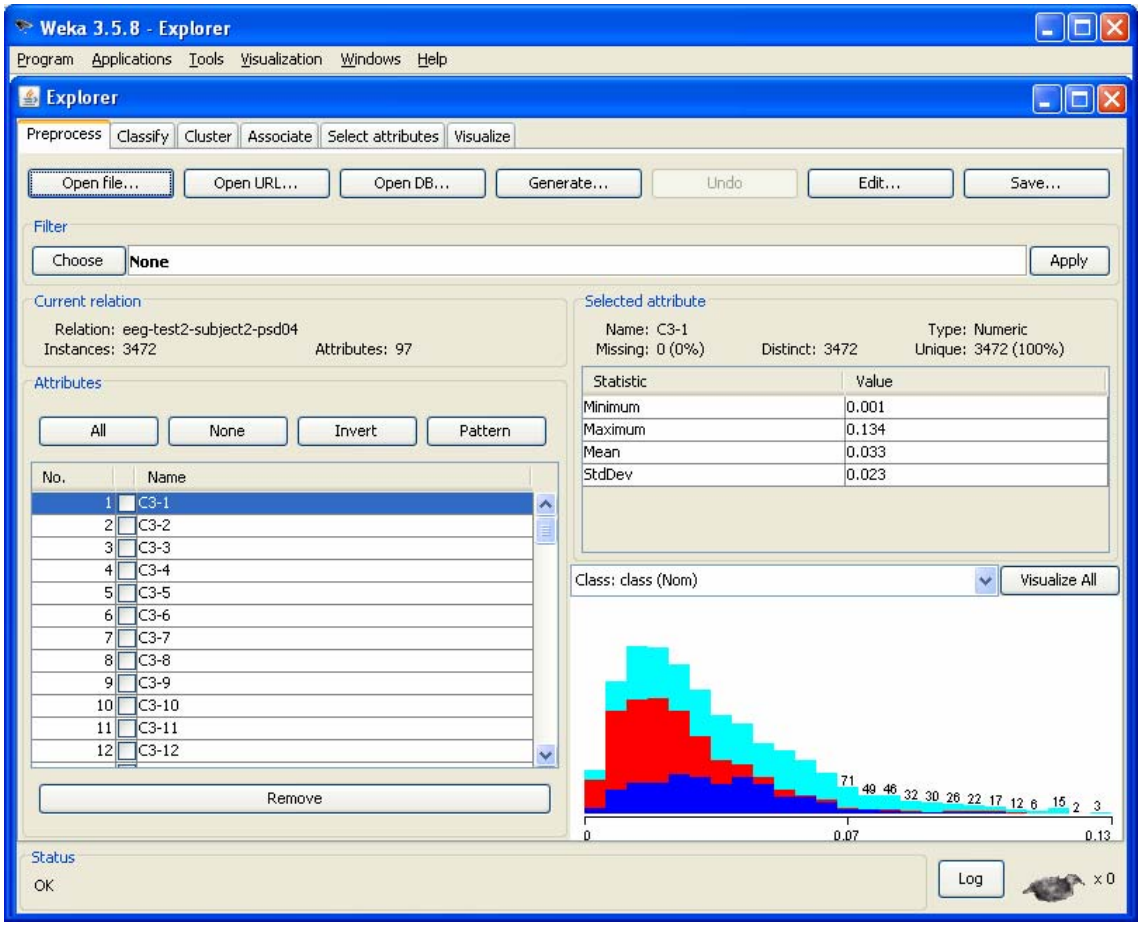

Ilustración 5.4 – Pestaña Preprocesado de la interfaz Explorador de *Weka*

Una vez cargadas las instancias, el usuario puede editarlas y eliminar atributos explícitamente, así como aplicarles filtros. Los filtros se dividen en supervisados y no supervisados y, a su vez, en filtros de instancia o de atributo. Los filtros más utilizados son los no supervisados, a modo de ejemplo se expondrán algunos de ellos:

- Filtros no supervisados de atributo:

    - *Discretize*: Discretiza un rango de atributos numéricos del conjunto de datos por atributos nominales.

    - *Remove*: Elimina un rango de atributos del conjunto de datos.

    - *ReplaceMissingValues*: Reemplaza todos los valores desconocidos por valores nominales o numéricos conservando las medias y las modas del conjunto de datos de entrenamiento.

- Filtros no supervisados de instancia:

    - *RemovePercentage*: Elimina un determinado porcentaje de instancias del conjunto de datos.

    - *RemoveWithValues*: Elimina las instancias acordes al valor de uno de sus atributos.

    - *Normalize*: Normaliza instancias considerando solo atributos numéricos.

  Durante el proceso de preprocesado de los datos puede visualizarse gráficamente para cada atributo (o para todos) cuántos valores distintos tiene, cuáles son sus valores mínimo y máximo, su media… Una vez preprocesado el conjunto de datos se tiene la opción de guardarlo en cualquiera de los formatos aceptados por *Weka* y comentados anteriormente o pulsar en cualquiera de las demás pestañas para seguir trabajando con los datos.

- **Clasificación**: Corresponde con la pestaña *Classify*. Desde ésta puede clasificarse y aprender de cualquiera de los atributos, es decir, cualquier atributo puede ser seleccionado como atributo clase.

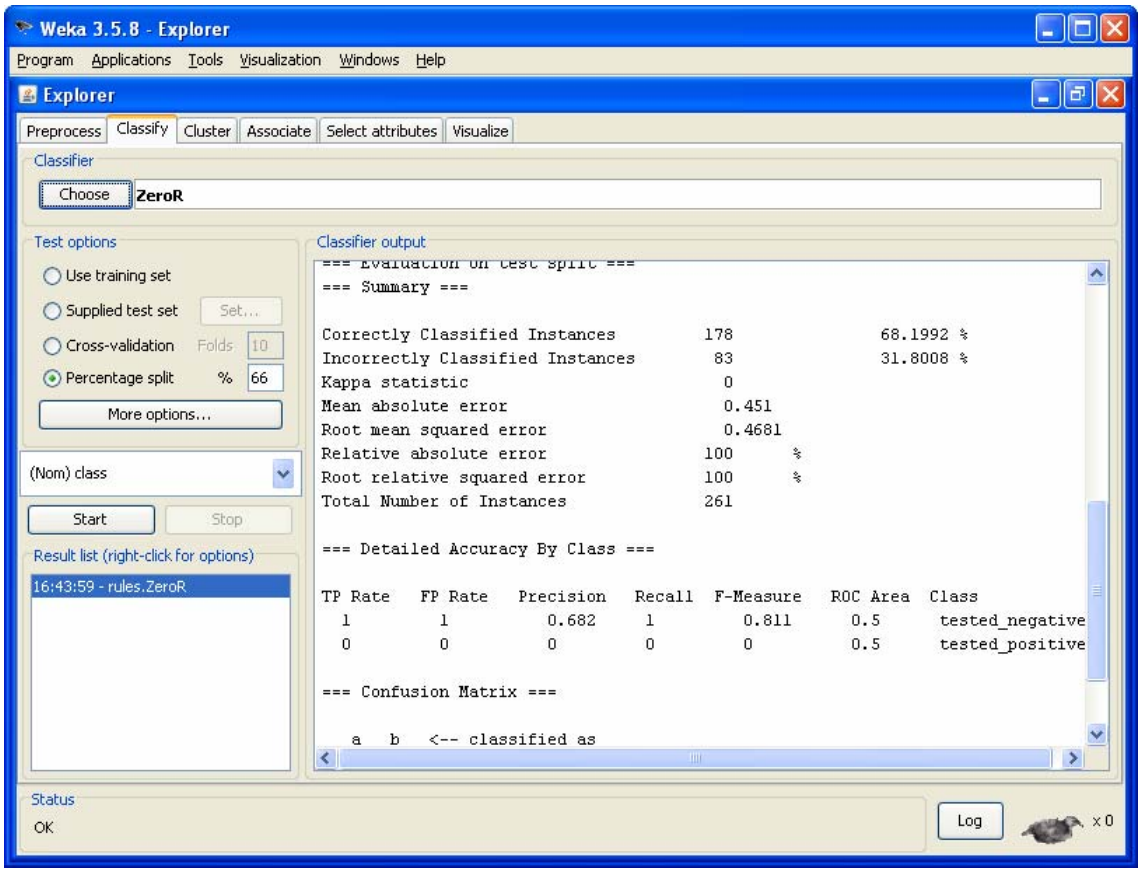

Ilustración 5.5 – Pestaña Clasificación de la interfaz Explorador de *Weka*

Pueden seleccionarse cuatro modos de test:

- **Use training set**: El conjunto de datos en su totalidad es utilizado para entrenar al clasificador y como posterior conjunto de test.

- **Supplied test set**: El conjunto de datos en su totalidad es utilizado para entrenar al clasificador y el conjunto proporcionado adicionalmente es utilizado como conjunto de test.

- **Cross-validation**: El conjunto de datos es utilizado como training y test siguiendo el esquema de validación cruzada, es decir, el conjunto de datos completo es troceado en un número de partes igual al número de hojas especificado y, durante un número de veces igual al número de hojas, un trozo hace de conjunto de test y los restantes de conjunto de training, variando el trozo que actúa como conjunto de test, de tal modo que cada instancia del conjunto de datos haya efectuado de test a lo sumo una vez.

- **Percentage Split**: En este caso se especifica un porcentaje en el intervalo (0, 100), de tal forma que un porcentaje de las instancias igual al especificado hace de training y las restante de test. Haciendo uso de una de las opciones adicionales proporcionadas por *Weka*, se puede explicitar que se mantenga el orden de las instancias al hacer la división en training y test.

Una vez seleccionado el atributo clase y el modo de evaluación, el algoritmo de clasificación a seleccionar puede ser de los siguientes tipos:

- **Bayes:** Utilizan el teorema de Bayes de una u otra manera, los algoritmos que se engloban en esta categoría son los siguientes:
  - *AODE*
  - *AODEsr*
  - *BayesianLogisticRegression*
  - *BayesNet*
  - *ComplementNaiveBayes*
  - *DMNBtext*
  - *HNB*
  - *NaiveBayes*
  - *NaiveBayesMultinomial*
  - *NaiveBayesMultinomialUpdateable*
  - *NaiveBayesSimple*
  - *NaiveBayesUpdateable*
  - *WAODE*

- **Functions**: Se refiere por funciones a aquellos algoritmos que podrían ser descritos utilizando ecuaciones matemáticas. Los algoritmos que se engloban en esta categoría (se excluye, entre otros, *Naive Bayes* por pertenecer a la categoría Bayes) son los siguientes: *GaussianProcesses, IsotonicRegression, LeastMedSq, LibSVM, LinearRegression, Logistic, MultilayerPerceptron,*

PaceRegression, PLSClassifier, RBFNetwork, SimpleLinearRegression, SimpleLogistic, SMO, SMOreg, SVMreg, VotedPerceptron y Winnow.

- **Lazy:** Se denominan perezosos y los algoritmos que almacenan, pero no trabajan con las instancias de entrenamiento, hasta el momento de realizar la clasificación de otras instancias. Los algoritmos que se engloban en esta categoría son los siguientes:
  - *IB1*
  - *IBk*
  - *KStar*
  - *LBR*
  - *LWL*

- **Meta:** Se denominan meta-algoritmos a aquellos algoritmos que utilizan conjuntamente los resultados de otros algoritmos (algoritmos base) para proporcionar una respuesta más precisa. Los algoritmos que se engloban en esta categoría son los siguientes: *AdaBoostM1, AdditiveRegression, AttributeSelectedClassifier, Bagging, ClassificationViaClustering, ClassificationViaRegression, CostSensitiveClassifier, CVParameterSelection, Dagging, Decorate, END, EnsembleSelection, FilteredClassifier, Grading, GridSearch, LogitBoost, MetaCost, MultiBoostAB, MultiClassClassifier, MultiScheme, rdinalClassClassifier, RacedIncrementalLogitBoost, RandomCommittee, RandomSubSpace, RegressionByDiscretization, Stacking, StackingC, ThresholdSelector y Vote. A su vez, en el paquete nestedDichotomies encontramos los siguientes: ClassBalancedND, DataNearBalancedND y ND.*

- **Mi:** Los algoritmos que se engloban en esta categoría son los siguientes: *CitationKNN, MDD, MIBoost, MIDD, MIEMDD, MILR, MINND, MIOptimalBall, MISMO, MISVM, MIWrapper, SimpleMI, TLD y TLDSimple.*

- **Misc:** Los algoritmos que se engloban en esta categoría son los siguientes: *FLR, HyperPipes, MinMaxExtension, OLM, OSDL, SerializedClassifier y VFI.*

- **Trees:** Los algoritmos que se engloban en esta categoría son aquellos que construyen árboles con los datos obtenidos de las instancias de entrenamiento y los utilizan a la hora de clasificar y son los siguientes:
  - *ADTree*
  - *BFTree*
  - *DecisionStump*
  - *FT*
  - *Id3*
  - *J48*
  - *J48graft*

- *LMT*
- *M5P*
- *NBTree*
- *RandomForest*
- *RandomTree*
- *REPTree*
- *SimpleCart*
- *UserClassifier*

o **Rules:** Los algoritmos que se engloban en esta categoría son aquellos que proporcionan modelos como conjuntos de reglas y son los siguientes: *ConjunctiveRule, DecisionTable, DTNB, JRip, M5Rules, NNge, OneR, PART, Prism, Ridor y ZeroR.*

- **Clustering**: Permite aplicar al conjunto de datos preprocesado diferentes algoritmos de agrupamiento.

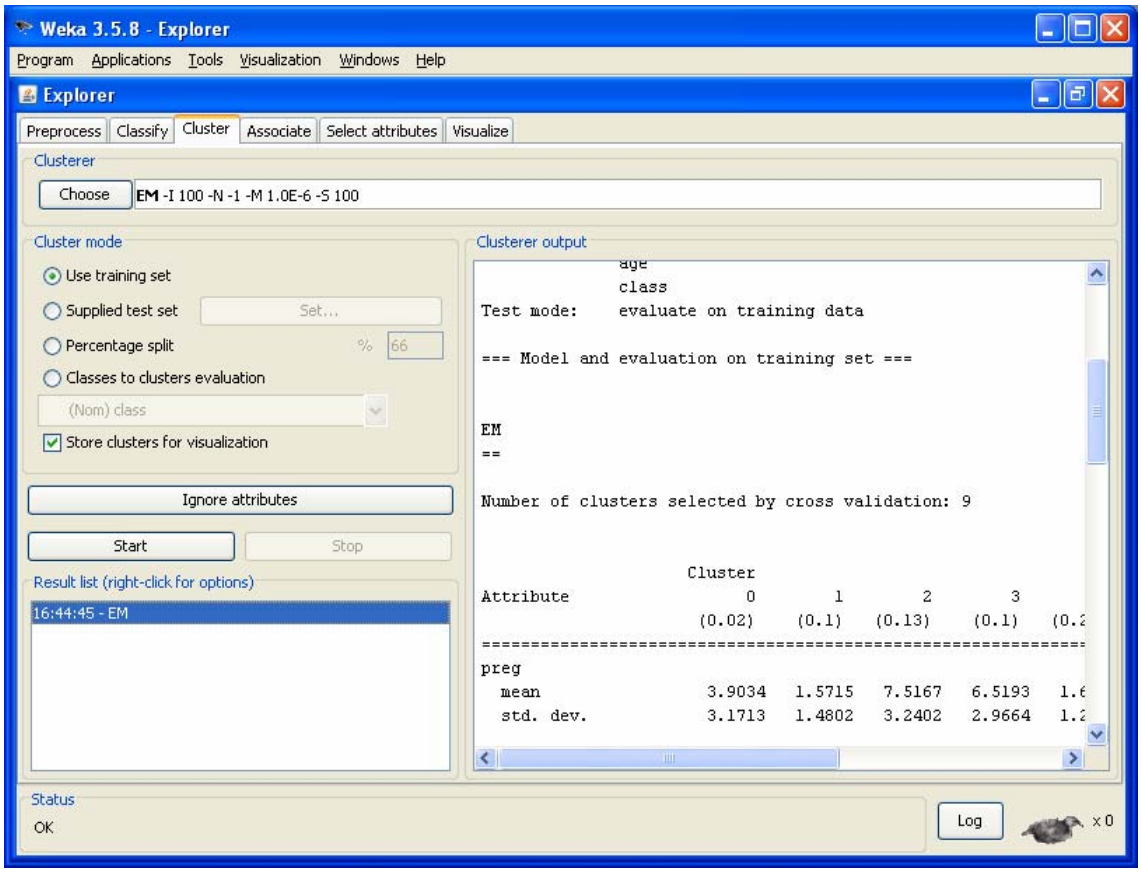

Ilustración 5.6 – Pestaña Cluster de la interfaz Explorador de *Weka*

Los algoritmos disponibles son los siguientes:

- *CLOPE*
- *Cobweb*
- *DBScan*
- *EM*
- *FarthestFirst*

- *FilteredClusterer*
- *MakeDensityBasedClusterer*
- *OPTICS*
- *sIB*
- *SimpleKMeans*
- *XMeans*

- **Búsqueda de Asociaciones**: Permite aplicar al conjunto de datos preprocesado diferentes algoritmos orientados a buscar asociaciones entre los datos.

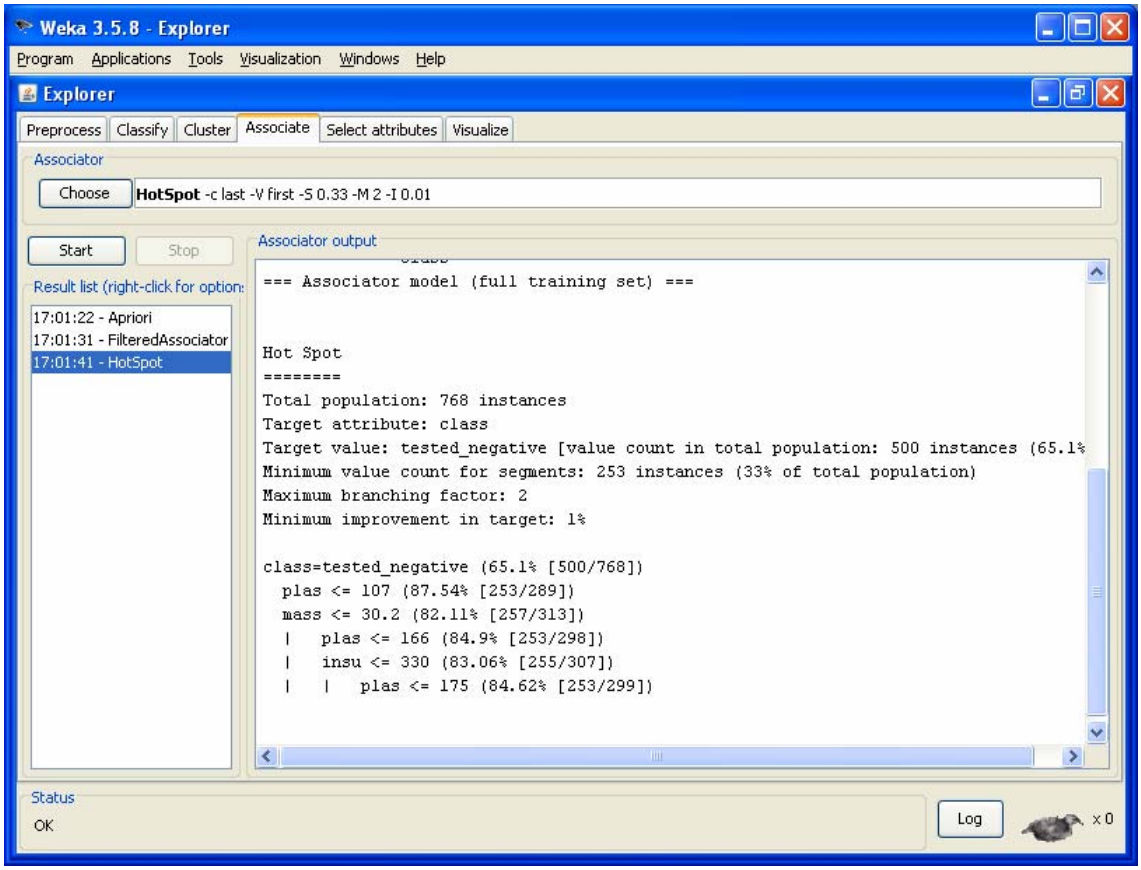

Ilustración 5.7 – Pestaña Associate de la interfaz Explorador de *Weka*

Los algoritmos disponibles son los siguientes:

- *Apriori*
- *FilteredAssociator*
- *GeneralizedSequentialPatterns*
- *HotSpot*
- *PredictiveApriori*
- *Tertius*

- **Selección de atributos**: Permite aplicar al conjunto de datos preprocesado diferentes algoritmos para identificar aquellos atributos que tienen más peso a la hora de clasificar las instancias.

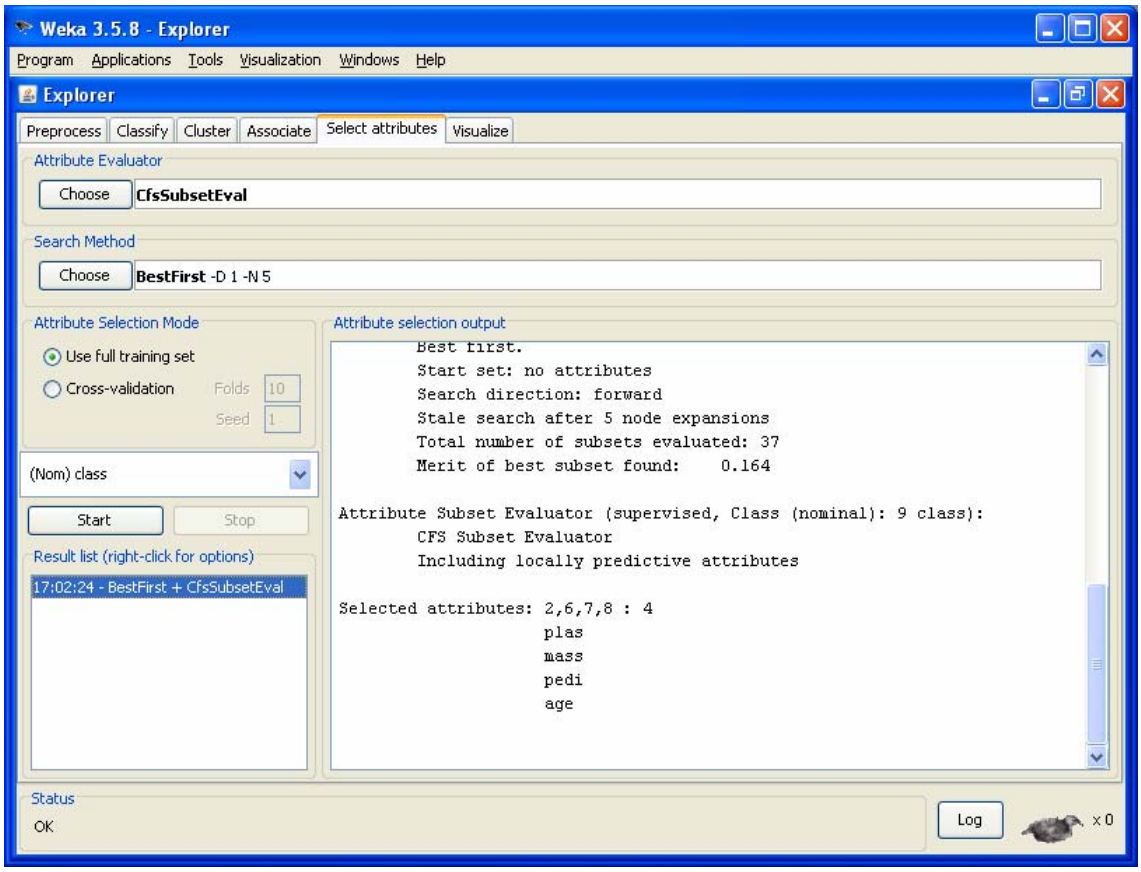

Ilustración 5.8 – Pestaña Selección Atributos de la interfaz Explorador de *Weka*

Como puede verse, para seleccionar atributos en un conjunto de datos ha de seleccionarse un algoritmo de selección de atributos y un algoritmo para moverse por el espacio de búsqueda. Los algoritmos de selección de atributos disponibles son los siguientes:

- *ASEvaluation*
- *CfsSubsetEval*
- *ChiSquaredAttributeEval*
- *ClassifierSubsetEval*
- *ConsistencySubsetEval*
- *CostSensitiveAttributeEval*
- *CostSensitiveSubsetEval*
- *FilteredAttributeEval*
- *FilteredSubsetEval*
- *GainRatioAttributeEval*
- *InfoGainAttributeEval*
- *LatentSemanticAnalysis*
- *OneRAttributeEval*
- *PrincipalComponents*
- *ReliefFAttributeEval*
- *SVMAttributeEval*
- *SymmetricalUncertAttributeEval*
- *SymmetricalUncertAttributeSetEval*
- *WrapperSubsetEval*

Los métodos de búsqueda disponibles son los siguientes:

- *BestFirst*
- *ExhaustiveSearch*
- *FCBFSearch*
- *GeneticSearch*
- *GreedyStepwise*
- *LinearForwardSelection*
- *RaceSearch*
- *RandomSearch*
- *Ranker*
- *RankSearch*
- *SubsetSizeForwardSelection*

- **Visualización de datos**: Permite visualizar la distribución de los datos de acuerdo con los valores de dos atributos cualesquiera (gráficas bidimensionales).

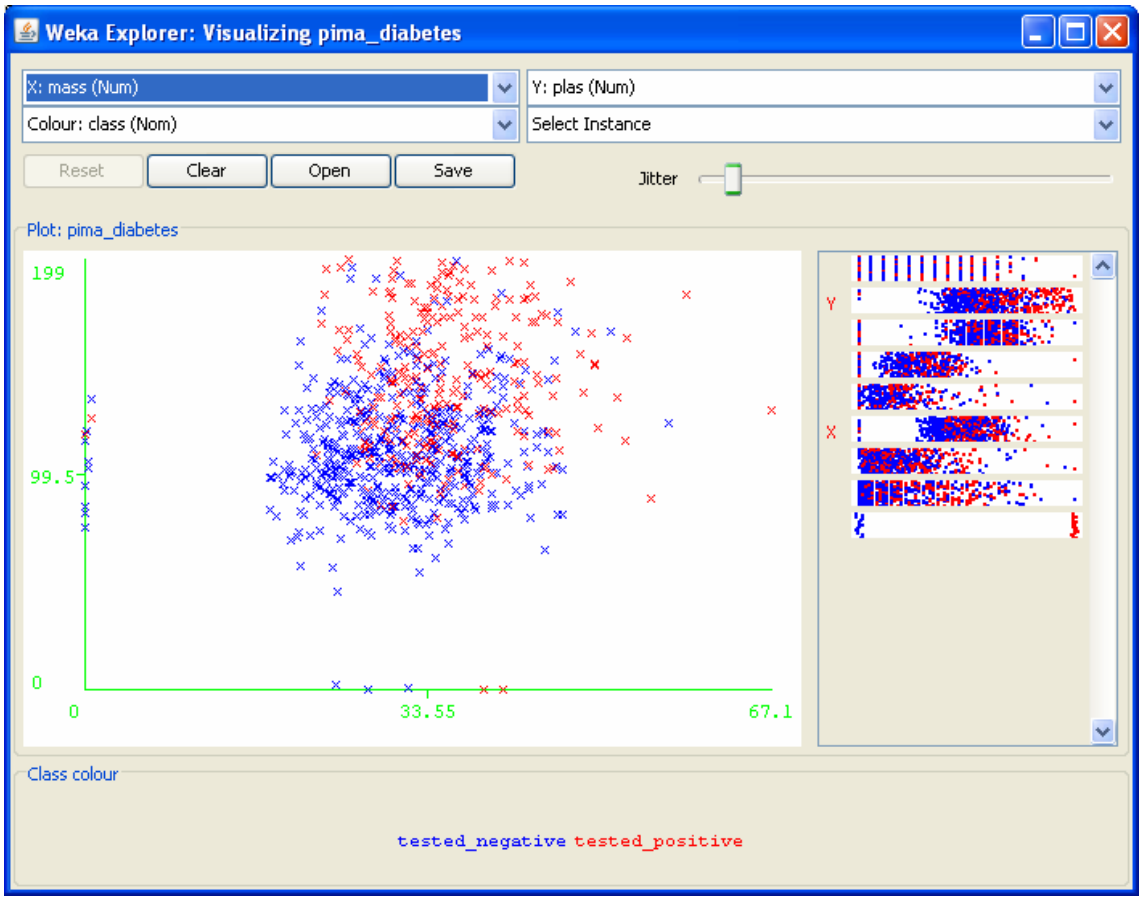

Ilustración 5.9 – Visualización bidimensional de dos atributos

Como puede observarse, los datos aparecen coloreados en función de la clase a la que pertenezca su instancia. Para una mejor visualización puede utilizarse la barra de desplazamiento, *Jitter*, que va moviendo los datos aleatoriamente desde su posición inicial.

Al hacer clic sobre esta pestaña, se muestran gráficamente las distribuciones de todos los atributos dos a dos. Por tanto, si n es el número de atributos (incluida la clase), habrá n^2 gráficas bidimensionales. Esta opción es muy útil para detectar asociaciones y relaciones rápidamente de manera gráfica. Se dispone de varias opciones realmente útiles:

- *PlotSize*: Tamaño de las gráficas bidimensionales donde se muestran los datos dependiendo de los valores de dos de sus atributos.
- *PointSize*: Cada instancia es representada con un punto en la gráfica. Con esta opción se puede variar su tamaño.
- *Jitter*: Como se explicó previamente, va moviendo los datos aleatoriamente desde su posición inicial. Es útil cuando los valores de los atributos son muy similares.
- *Select Attributes*: Permite seleccionar/deseleccionar los atributos que le interese estén en las gráficas.
- *Subsample %*: Permite especificar un porcentaje de las instancias del conjunto de datos que serán utilizadas para ser representadas en las gráficas.

A continuación se muestra un ejemplo, donde los datos han sido desplazados con Jitter, los tamaños de los puntos han sido ampliados, se han seleccionado solamente tres atributos y se emplean sólo el 60% de las instancias del conjunto de test:

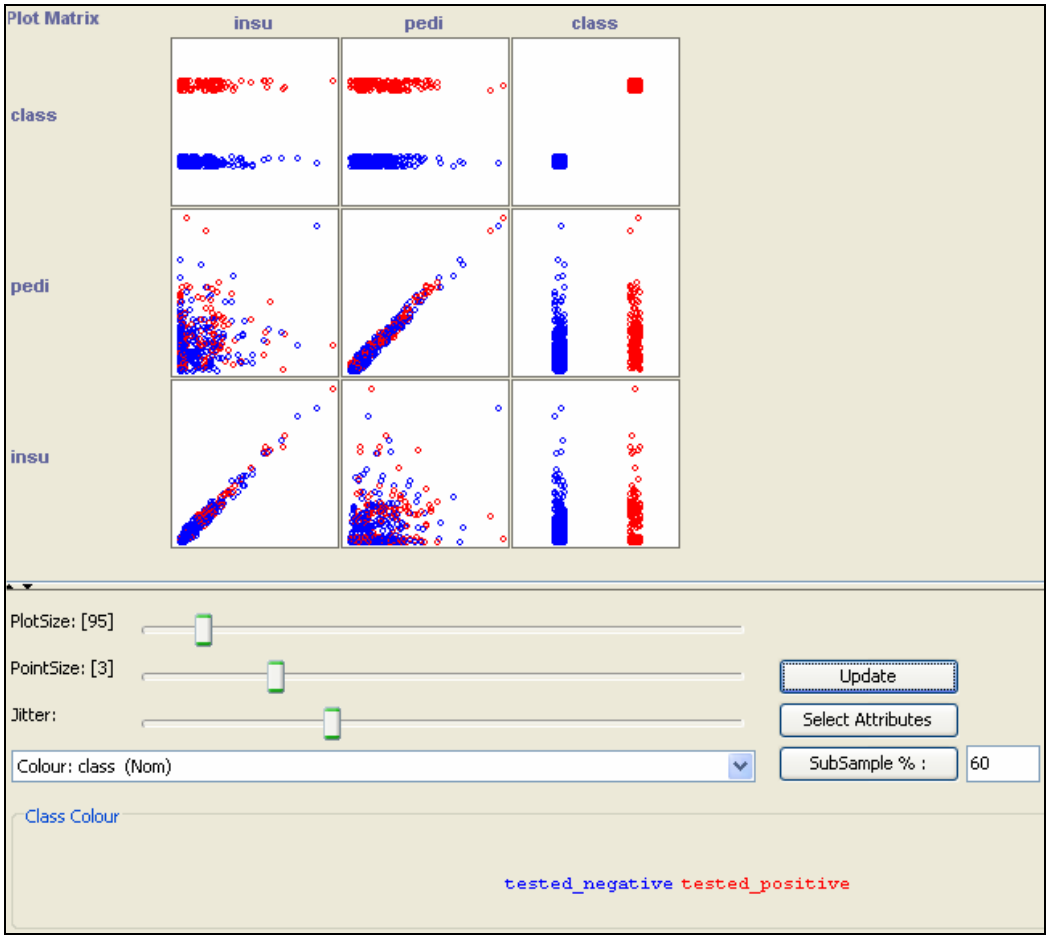

Ilustración 5.10 – Visualización de los valores de los atributos confrontados

5.3.2 Experimenter

Permite aplicar un conjunto de algoritmos sobre uno o más conjuntos de datos y un posterior análisis estadístico sobre ellos para determinar cuán buenos son. El experimentador de *Weka* resulta ser una gran ayuda para llevar a cabo experimentos a gran escala.

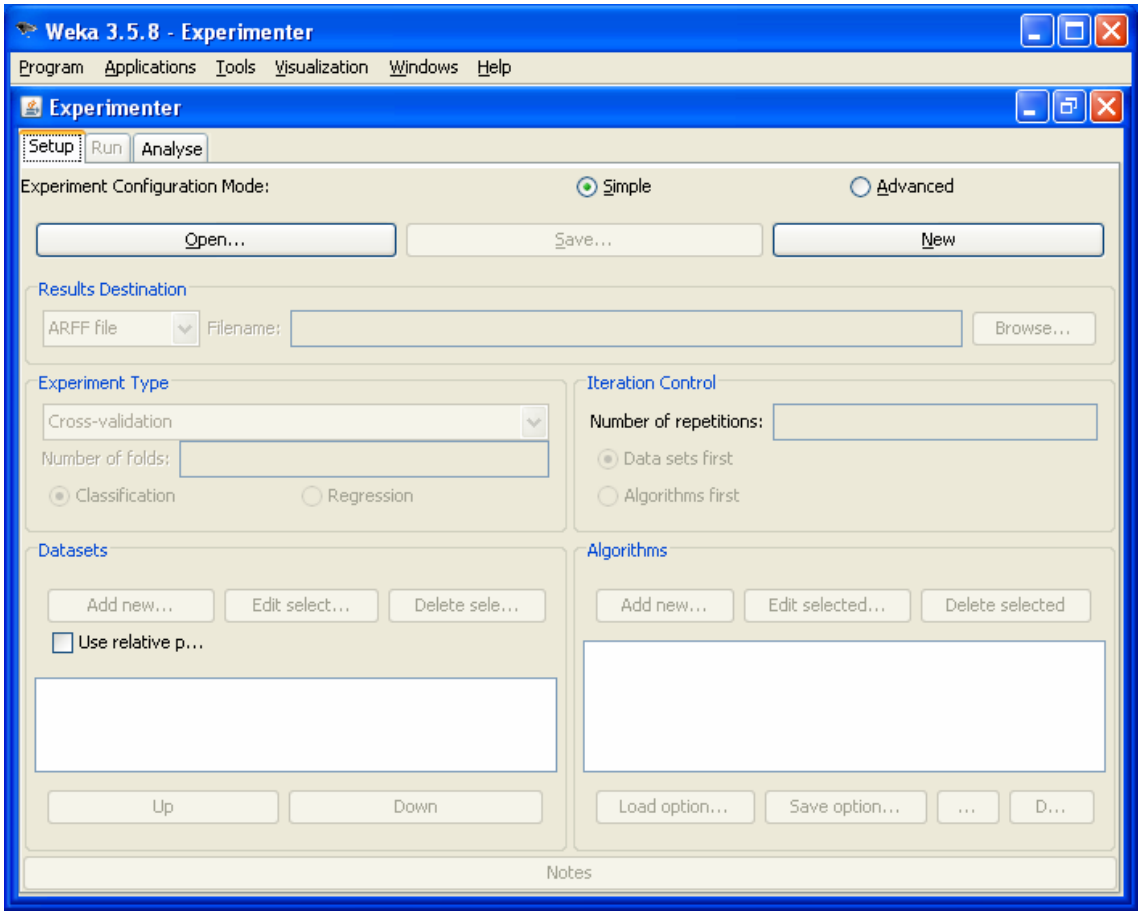

Ilustración 5.11 – Modo de configuración simple de experimentador

Una vez abierto el experimentador visualizaremos una interfaz tal como se muestra en la ilustración. Como puede observarse, hay tres pestañas:

- *Setup*: Permite configurar el experimento. Proporciona dos interfaces de configuración: simple y avanzada.

  En modo de configuración simple es necesario definir un fichero en el que guardar los resultados del experimento. Posteriormente, se elige un modo de test y se cargan los conjuntos de datos sobre los que se realizarán experimentos.

  Hay disponibles tres modos de test:

- Validación de cruzada
- Porcentaje de la población (*split*) tomada de manera ordenada
- Porcentaje de la población (*split*) tomada aleatoriamente

Por último se seleccionarán los algoritmos que se aplicarán a los conjuntos de datos, indicando el número de repeticiones y si se itera en primer lugar por conjuntos de datos o algoritmos.

Por ejemplo, si el usuario decide iterar en primer lugar por algoritmos, el experimentador realizará el experimento con el siguiente bucle:

```
Para cada algoritmo
  Para cada conjunto de datos
    Para el número de repeticiones indicado
       Aplica el algoritmo al conjunto de datos
       con el modo de test especificado
```

El modo de configuración avanzado ofrece, entre otras cosas, la posibilidad de distribuir el cómputo entre varios ordenadores.

Ambos modos de configuración ofrecen la posibilidad de guardar y cargar experimentos.

- *Run*: Desde esta se lleva a cabo la ejecución del experimento. Sólo tiene dos opciones: empezar el experimento y abortarlo.

- *Analyse*: Permite realizar un análisis estadístico de los resultados obtenidos.

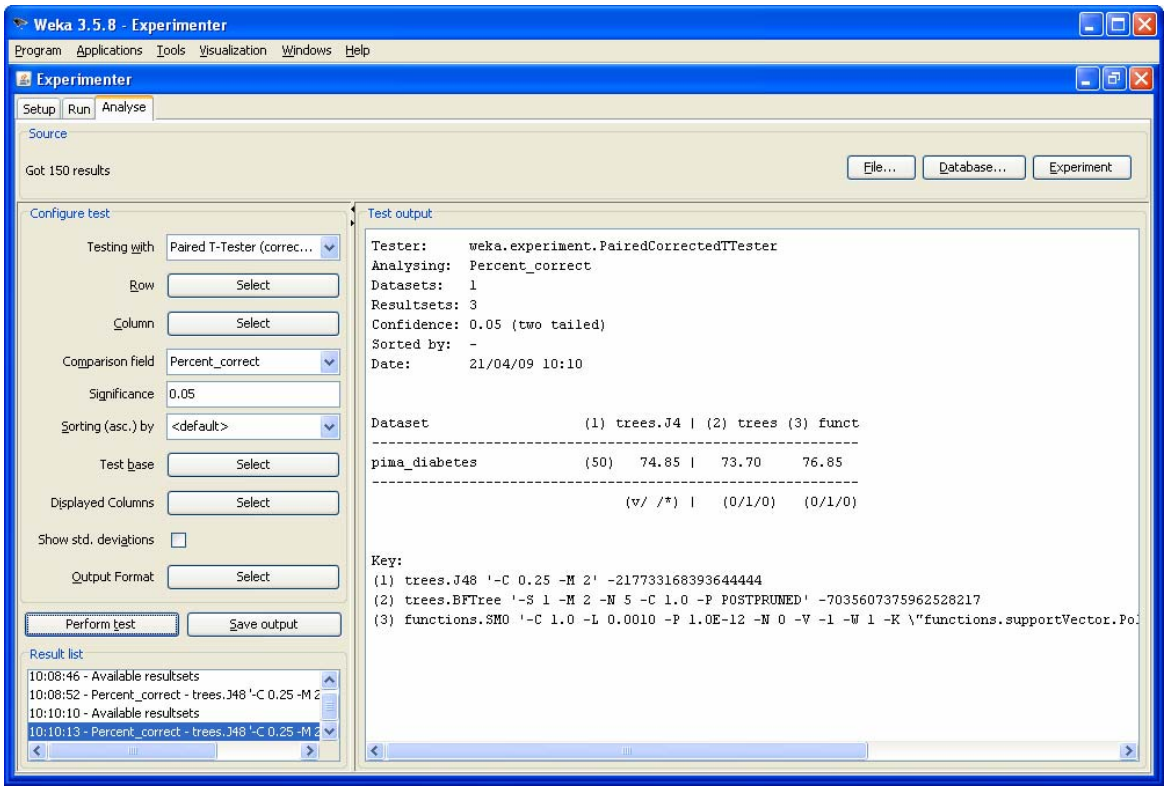

Ilustración 5.12 – Pestaña de análisis del experimentador de *Weka*

Es muy sencillo comparar los resultados de los diferentes algoritmos implementados. En primer lugar se carga el fichero con los resultados del experimento, se configura el test y se pulsa el botón para llevarlo a cabo. Puede definirse el tamaño de la matriz de comparación, nivel de significación…

5.3.3 KnowledgeFlow

Esta opción de *Weka* permite al usuario definir su experimento de manera gráfica. Es una interfaz equivalente al explorador, su funcionalidad es la misma, por tanto, sólo resulta útil para quienes piensen mejor en término de cómo fluyen los datos…

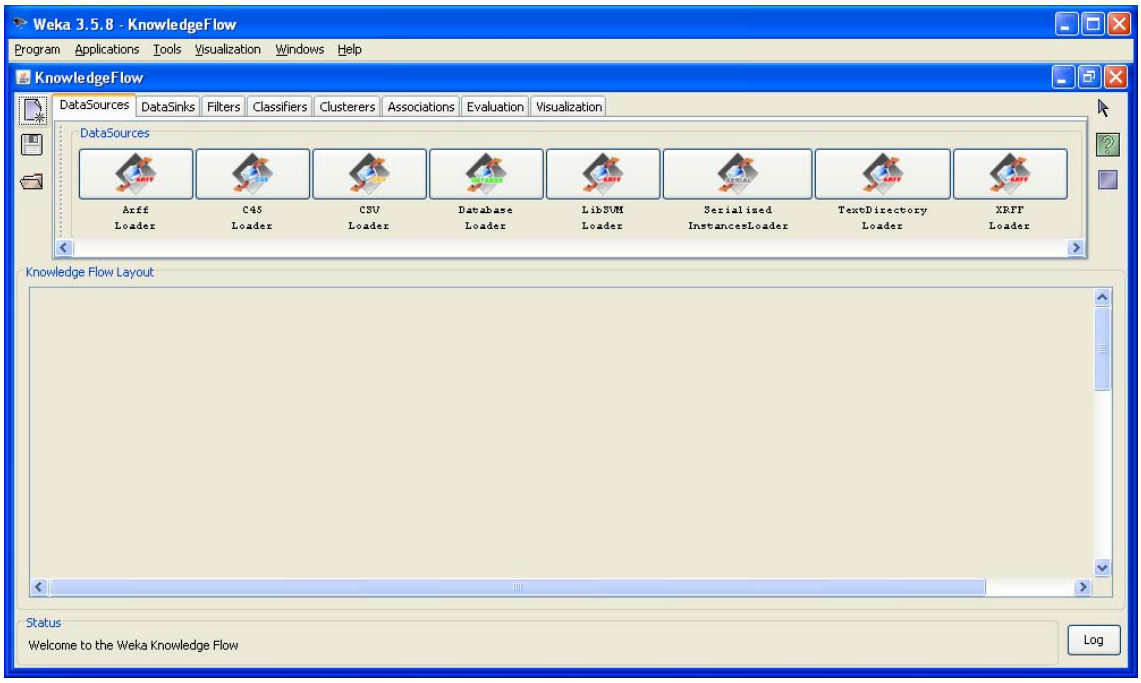

Ilustración 5.13 – *KnowledgeFlow* de *Weka*

Tiene definidas pestañas en las que se pueden seleccionar elementos gráficos, tales como: orígenes de datos, filtros, clasificadores, elementos de visualización, elementos de evaluación…

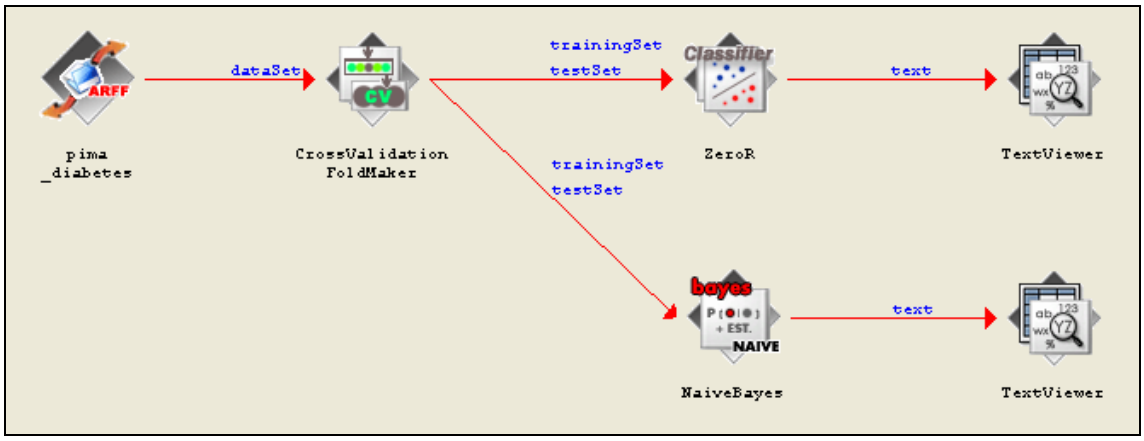

Ilustración 5.14 – Ejemplo de experimento en *KnowledgeFlow*

Como puede verse en la ilustración, el experimento realiza los siguientes pasos: el elemento *ArffLoader* carga un conjunto de datos, se realiza una validación cruzada con los algoritmos *ZeroR* y *NaiveBayes* y, en último lugar, se colocan los elementos *TextViewer* para poder visualizar los resultados.

Una vez creado el experimento, para ejecutarlo se hace clic con el botón derecho sobre el elemento *ArffLoader* y se selección la acción: *StartLoading* (comenzar la carga). Una vez ha concluido, seleccionando (clic con el botón derecho) la acción *Show results* (mostrar resultados) de los elementos *TextViewer* se visualizan los resultados de la aplicación de cada algoritmo.

5.3.4 SimpleCLI

Abreviación de Simple Client (Cliente Simple), permite utilizar *Weka* mediante mandatos por línea de comandos. Ofrece la misma funcionalidad que *Weka* ofrece en su interfaz, pero apenas se emplea porque su poco intuitivo uso exige tener un conocimiento avanzado de la aplicación. Cuando se invoca aparece la ventana mostrada en la ilustración 5.15:

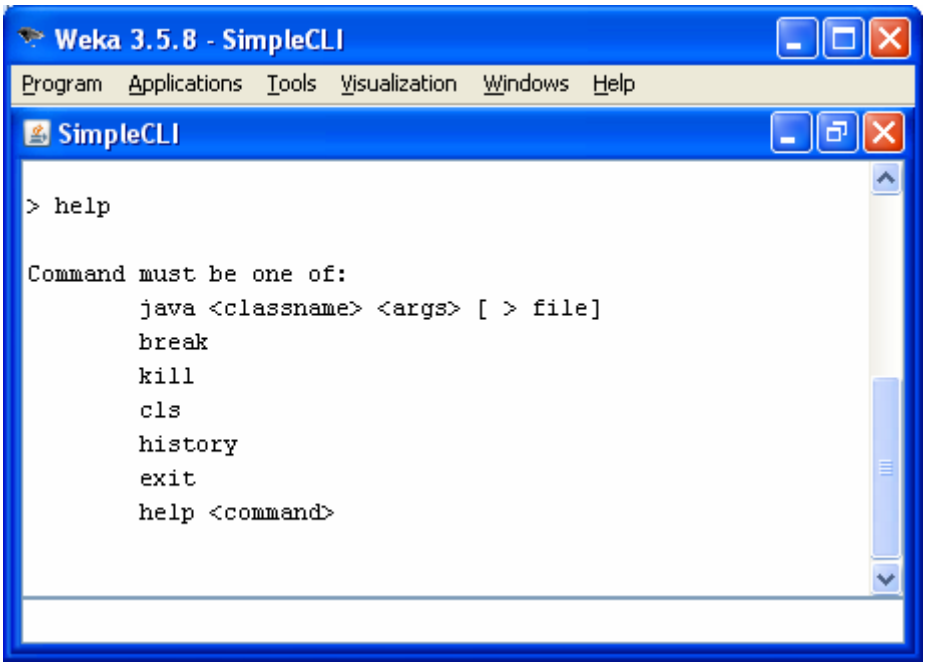

Ilustración 5.15 – *SimpleCLI* en *Weka*

Los comandos disponibles son:

- *java*: Seguido del nombre de la clase y sus argumentos, invoca el método *main* de dicha clase con los argumentos especificados, obligatorios en la gran mayoría de los casos.
- *break*: Detiene su ejecución.
- *kill*: Aborta su ejecución.
- *cls*: Limpia la pantalla.
- *history*: Muestra por pantalla la historia de comandos.
- *exit*: Se cierra la aplicación SimpleCLI.
- *help*: Sólo o seguida de un comando, ofrece ayuda relacionada.

5.4 Estructura de Weka

Weka se compone de 1149 clases distribuidas en los paquetes:

- *associations*
- *attributeSelection*
- *classifiers*
- *clusterers*
- *core*
- *datagenerators*
- *estimators*
- *experiment*
- *filters*
- *gui*

Otro importante fichero, fuera de estos paquetes, es *build.xml*, un archivo que contiene reglas de compilación.

En los siguientes apartados se describirá detalladamente cada uno de los paquetes.

5.4.1 Paquete associations

El paquete *associations* contiene todas las clases necesarias para poder trabajar con la pestaña de asociación del explorador de *Weka*. Estas clases tienen implementados los algoritmos disponibles por el usuario: *Apriori*, *HotSpot*… Además, contiene el paquete *tertius*, que contiene las clases para implementar el método *tertius*, un sistema para el descubrimiento de reglas en lógica de primer orden.

5.4.2 Paquete attributeSelection

El paquete *attributeSelection* contiene clases que implementan los algoritmos de selección de atributos y los métodos de búsqueda disponibles en la pestaña de selección de atributos del explorador de *Weka*. Define seis interfaces que sirven para algoritmos que evalúan atributos individualmente o subconjuntos de atributos, para aquellos que producen un ranking de atributos…

5.4.3 Paquete classifiers

El paquete *classifiers* es aquel donde están implementados todos los clasificadores disponibles en *Weka*. Contiene la clase *Classifier.java*, una clase abstracta que, directa o indirectamente, han de heredar obligatoriamente todos los clasificadores implementados en *Weka*. Otra opción sería heredar los nuevos clasificadores de las clases

MultipleClassifiersCombiner.java, IteratedSingleClassifierEnhancer.java, SingleClassifierEnhancer.java… que heredan de *Classifier.java* y amplían sus características.

La clase *Classifier.java* define el atributo *m_Debug* (depuración), lo muestra en el cuadro de opciones del clasificador y actualiza su valor con el valor seleccionado por el usuario. Además, define métodos para ejecutar el clasificador, para construirlo, para clasificar instancias…

Cuenta con las siguientes interfaces:

- *IntervalEstimator*: Interfaz para clasificadores que pueden trabajar con intervalos de confianza a la hora de predecir a qué clase pertenece una instancia. Utilizada únicamente por el clasificador *GuassianProcesses*.
- *IterativeClassifier*: Interfaz para clasificadores que pueden construir modelos de complejidad creciente, conforme va procesando instancias. Utilizada únicamente por el clasificador *ADTree*.
- *Sourcable*: Interfaz para clasificadores que pueden ser convertidos a código Java. Utilizado por los clasificadores *Id3, OneR, ZeroR*… Cuando se ejecuta alguno de estos algoritmos habiendo seleccionado la opción *Output source code* en el botón de más opciones de la pestaña de clasificación del explorador de *Weka*, se muestra por consola el código fuente de una copia simplificada del clasificador.
- *UpdateableClassifier*: Interfaz para modelos de clasificación incremental que pueden aprender utilizando una instancia tras otra. Es utilizada por los clasificadores *NaiveBayesUpdateable, IB1, IBk, KStar*…

El paquete *classifiers* contiene a su vez los siguientes paquetes:

- *bayes*: Contiene implementaciones de clasificadores basados en el teorema de Bayes.
- *evaluation*: Contiene clases con funcionalidades adicionales, no algoritmos en sí. Sus clases permiten trabajar con matrices de coste, predicciones…
- *functions*: Contiene implementaciones de algoritmos basados en funciones.
- *lazy*: Contiene implementaciones de técnicas perezosas.
- *meta*: Contiene implementaciones de meta-clasificadores.
- *mi:* Contiene algoritmos para clasificación multi-instancia.
- *misc*: Contiene implementaciones de clasificadores que no se encuentran en ninguna de las restantes categorías.
- *rules*: Contiene implementaciones de clasificadores que utilizan reglas de decisión.
- *trees*: Contiene implementaciones de clasificadores que utilizan árboles.
- *xml*: Contiene una clase para serializar y deserializar instancias en XML.

5.4.4 Paquete clusterers

El paquete *clusterers* contiene las clases necesarias para poder trabajar con la pestaña *Cluster* del explorador de *Weka*. En sus clases están definidos los once algoritmos disponibles (*Cobweb*, *EM*…) y cuenta con cuatro interfaces:

- *Clusterer*: Interfaz para los clusterers (agrupadores).
- *DensityBasedClusterer*: Interfaz para los clusterers que pueden calcular la densidad para una instancia dada.
- *NumberOfClusterersRequestable*: Interfaz para los clusterers que pueden generar un número de clusters determinado.
- *UpdateableClusterer*: Interfaz para modelos de incrementales de cluster que van aprendiendo conforme van recibiendo instancias.

5.4.5 Paquete core

Como su propio nombre indica, el paquete *core* es el núcleo de *Weka*. Exceptuando el paquete *classifiers*, es el paquete con mayor número de clases. En este paquete se encuentran las principales clases: clases para crear atributos, instancias, conjuntos de datos…

Las interfaces que se definen en este paquete tienen en común que pueden resultar útiles a algoritmos implementados en cualquiera de los restantes paquetes, en contraposición a las interfaces definidas en el resto de paquetes, que mayoritariamente son implementados por algoritmos de su paquete. Alguna de sus interfaces son las siguientes:

- *CapabilitiesHandler*: Las clases que implementen esta interfaz devuelven sus capacidades/habilidades con respecto a los conjuntos de datos que aceptan. Por ejemplo, si pueden trabajar con clases numéricas, atributos nominales…
- *Copyable*: Interfaz implementada por las clases que pueden producir copias de sus objetos.
- *DistanceFunction*: Interfaz para los algoritmos que puedan calcular la distancia entre dos instancias.
- *Drawable*: Útil para aquellas clases que puedan dibujar sus modelos (árboles…) como un grafo.
- *Matchable*: Interfaz para aquellas clases que utilicen árboles. Proporciona un método para transformarlo en modo prefijo.
- *MultiInstanceCapabilitiesHandler*: Equivalente a *CapabilitiesHandler* cuando se trabaja con multi-instancias[5].

[5] Una multi-instancia es una instancia en la que alguno de los atributos es definido como bag relational (bolsa relacional) y su/s valor/es puede/n tomar diferentes valores simultáneamente para una misma instancia.

- *OptionHandler*: Interfaz para los algoritmos que acepten opciones como parámetros. Define métodos para mostrar las opciones disponibles, para leer los valores introducidos por el usuario y para cargar dichos valores.
- *Randomizable*: Interfaz para las clases que necesiten trabajar con valores aleatorios, bien sea para barajar las instancias del conjunto de datos, bien porque el algoritmo necesite trabajar con datos aleatorios.
- *Summarizable*: Interfaz que proporciona un método que permite devolver una descripción textual del modelo resumida (en contraposición al método *toString()*). Por ejemplo, el clasificador PART implementa esta interfaz y sólo devuelve el número de reglas (descripción muy simplificada del modelo que ha construido).
- *TechnicalInformationHandler*: Interfaz que proporciona un método que permite especificar la información técnica referente al algoritmo. Por ejemplo, si es una publicación, el año de publicación los autores…
- *WeightedInstancesHandler*: Interfaz que deben implementar los algoritmos que trabajen con los pesos de las instancias. La interfaz no implementa ningún método y la implementación de la interfaz por parte de un algoritmo es un mero indicador.

Como se ha especificado anteriormente, en este paquete se encuentran las clases más importantes, en el sentido en que representan los objetos más básicos con que trabaja *Weka*: atributos, instancias, conjuntos de instancias (*datasets*), colas, matrices…

Las clases de este paquete más utilizadas en este proyecto (necesarias para comprenderlo) son *Attribute.java*, *Instance.java* e *Instances.java* y, por tanto, se explicarán detalladamente.

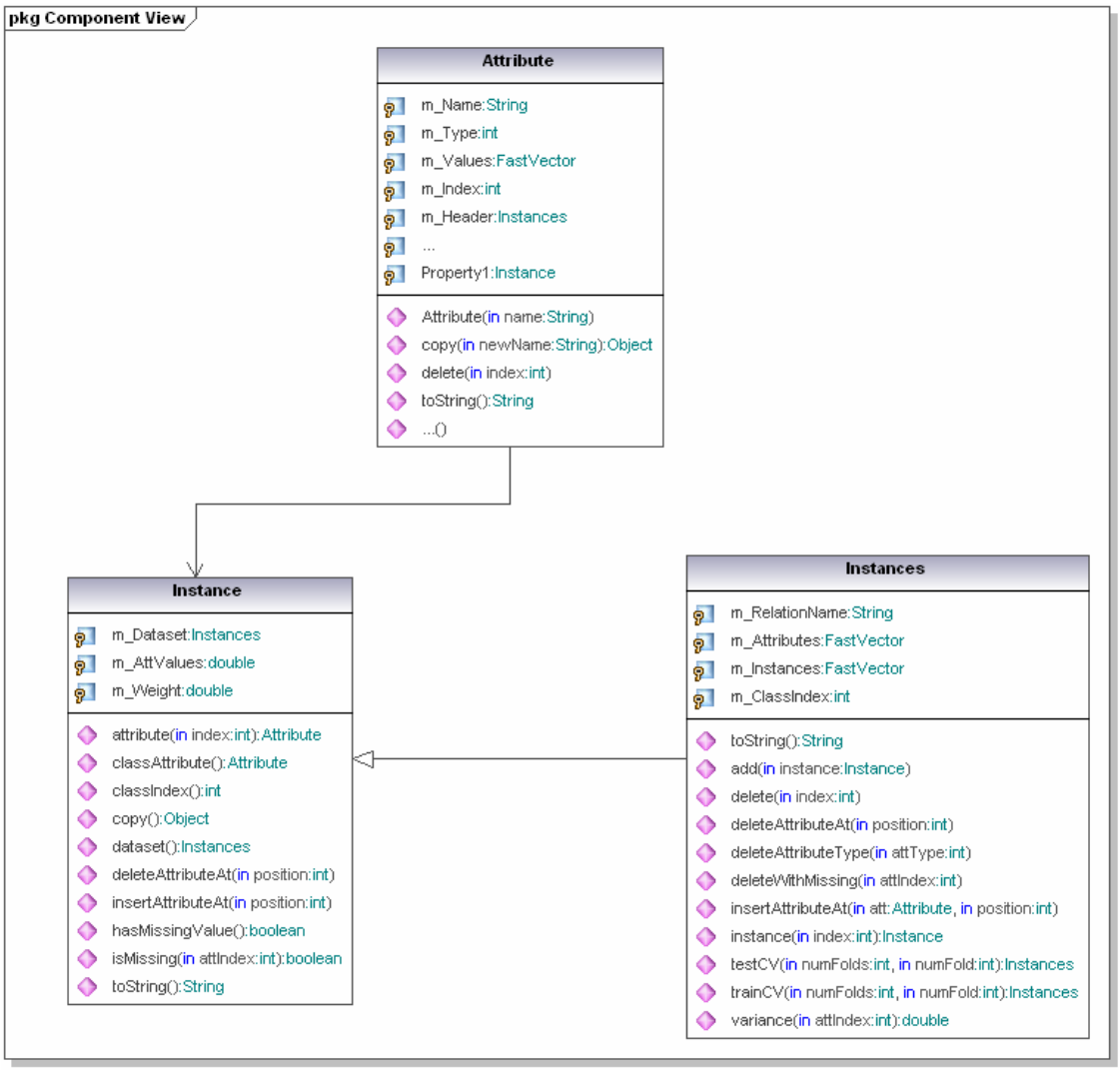

Ilustración 5.16 – Diagrama de clases para *Attribute*, *Instance* e *Instances*.

En el diagrama de clases sólo aparecen los métodos y atributos más importantes, refiriéndose a estos de tal manera puesto que logran proporcionar un conocimiento general de cómo las clases están implementadas sin hacer necesario mencionar a los restantes métodos y atributos, aunque algunos de ellos sí serán mencionados en la explicación textual que a continuación se hace de cada una de estas clases:

- **Clase Attribute**: La clase Attribute representa los atributos de las instancias. Define una serie de constantes, de tipo entero, para representar a cada uno de los posibles tipos que son: numérico, nominal, string, fecha o relacional. Sus más importantes atributos son:
  - *int m_Type*: Indica el tipo del atributo, fijará su valor dependiendo del valor que tomen las constantes mencionadas.
  - *String m_Name*: El nombre del atributo.
  - *FastVector m_Values*: Contiene los valores que puede tomar el atributo (inicializado sólo si es nominal, string o relacional).
  - *Instances m_Header*: Es una referencia al conjunto de datos (clase Instancias) a la que pertenece.

- *int m_Index*: El índice del atributo. Útil para trabajar con una instancia ya que contiene numerosos atributos, además, gracias a él se identifica el atributo clase.
- *double m_Weight*: El peso del atributo.

Entre sus numerosos métodos, existe una gran variedad de constructores, que permite crear atributos de cualquier tipo y con cualesquiera valores. Entre los restantes métodos, excluyendo accedentes y mutadores, los más importantes son:

- *Object copy()*: Realiza una copia del atributo. Es un método con el que ha de tenerse un gran cuidado, ya que la copia es idéntica, esto quiere decir que los atributos tienen el mismo nombre, índices… lo que puede originar un sinfín de problemas.
- *Enumeration enumerateValues()*: Devuelve los posibles valores que puede adoptar el atributo. Si el atributo es numérico o de tipo fecha el objeto devuelto será *null*.
- *String toString()*: Devuelve una descripción del atributo en formato ARFF.

- **Clase Instance**: La clase Instance representa cada una de las instancias del conjunto de datos. Sus atributos son los siguientes:
  - *Instances m_Dataset*: Conjunto de datos al que pertenece la instancia.
  - *double m_AttValues*: Es un array de valores reales que contienen los valores de todos los atributos de la instancia, incluida la clase.
  - *double m_Weight*: Peso de la instancia.

  Entre sus métodos más importantes, excluyendo constructores, accedentes y mutadores, se encuentran los siguientes:
  - *Attribute attribute(int index)*: Devuelve el atributo cuyo índice es *index*.
  - *Attribute classAttribute()*: Devuelve el atributo clase (aunque también podría obtenerse con el anterior método).
  - *Object copy()*: Devuelve una copia de la instancia.
  - *deleteAttributeAt(int position)*: Elimina el atributo que ocupa la posición *position*.
  - *insertAttributeAt(int position)*: Inserta un atributo en la posición *position*.
  - *boolean hasMissingValue()*: Comprueba si se conocen los valores de todos los atributos de la instancia.
  - *boolean isMissing(int attIndex)*: Equivalente al anterior, pero para un determinado atributo.

- **Clase Instances**: La clase Instances representa el conjunto de datos completo. Define los siguientes atributos:

- *String m_RelationName*: El nombre del conjunto de datos.
- *FastVector m_Attributes*: Información sobre los atributos.
- *FastVector m_Instances*: Información de las instancias pertenecientes al conjunto de datos.
- *int m_ClassIndex*: Índice del atributo clase.

Entre sus métodos más importantes, excluyendo constructores, accedentes y mutadores, se encuentran los siguientes:

- *add(Instance instance)*: Añade una instancia al conjunto de datos.
- *delete(int index)*: Elimina, del conjunto de datos, la instancia que ocupa la posición *index*.
- *deleteAttributeAt(int position)*: Elimina, del conjunto de datos, el atributo que ocupa la posición *position*.
- *deleteAttributeType(int attType)*: Elimina, del conjunto de datos, aquellos atributos de un determinado tipo.
- *deleteWithMissing(int attIndex)*: Elimina todas las instancias del conjunto de datos cuyo valor para el atributo con índice *attIndex* sea desconocido.
- *insertAttributeAt(Attribute att, int position)*: Inserta un atributo en una determinada posición.
- *Instance instance(int index)*: Devuelve la instancia que ocupa la posición *index*.
- *Instances testCV(int numFolds, int numFold)*: Devuelve el conjunto de datos con el que habría que hacer el test, cuando realizando validación cruzada con *numFolds* hojas, estamos por la hoja *numFold*.
- *Instances trainCV(int numFolds, int numFold)*: Equivalente al anterior pero devolviendo el conjunto de datos con el que habría que hacer el entrenamiento.
- *double variance(int attIndex)*: Calcula la varianza para un determinado atributo.

El paquete *core* contiene los siguientes paquetes:

- *converters*
- *mathematicalexpression*
- *matrix*
- *neighboursearch*
- *parser*
- *stemmers*
- *tokenizers*
- *xml*

Además, contiene tres ficheros de configuración:

- *version.txt*: Contiene la versión de *Weka*. Por ejemplo: 3-5-8
- *Capabilities.props*: Contiene información sobre capacidades/habilidades generales: si se puede hacer test con valores de clase desconocidos, para el mínimo número de instancias...
- *Copyright.props*: Contiene información del copyright, concretamente: entre qué años, propietario del copyright, dirección y url.

5.4.6 Paquete datagenerators

En la pestaña de preprocesado de datos del explorador de *Weka* se comentó que había un conjunto de algoritmos para generar conjuntos de instancias aleatoriamente. Esto es especialmente útil cuando no se dispone de conjunto de datos o se desea probar más exhaustivamente un algoritmo. El paquete *datagenerators* contiene las clases que implementan los algoritmos generadores de datos.

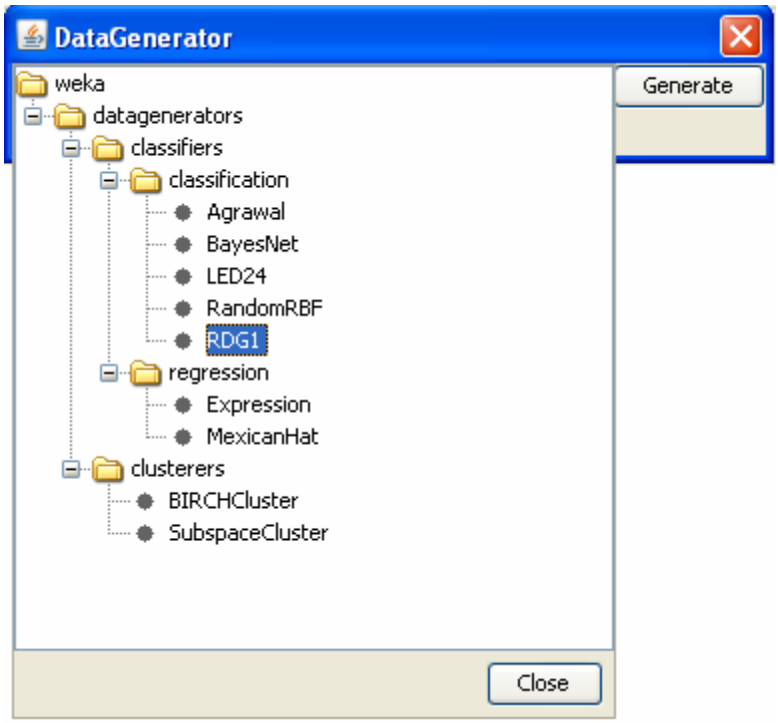

Ilustración 5.17 – Algoritmos generadores de datos

Como puede observarse, se han implementado diferentes generadores de datos, dependiendo de si se desean crear conjuntos de datos con instancias centradas en clusters (*clusterers*), con datos generados por clasificadores (*classifiers*) o fórmulas matemáticas (*regression*).

5.4.7 Paquete estimators

El paquete *estimators* contiene todas las clases necesarias para poder trabajar con estimadores en *Weka*. Cuenta con las interfaces *Estimator* y *ConditionalEstimator*, para estimadores de probabilidad y estimadores de probabilidad condicionada, respectivamente. Implementa el estimador de Poisson, el estimador normal…

5.4.8 Paquete experiment

El paquete *experiment* contiene todas las clases necesarias para poder trabajar con el Experimenter de *Weka*. Este paquete sirve para trabajar con cada uno de los experimentos (contiene las clases que representa el experimento en sí), tanto en local como en remoto.

Dispone de ficheros de configuración para establecer las propiedades de las conexiones con las bases de datos, en caso de que las hubiera. Existen ficheros de configuración diferentes para las diferentes posibilidades que ofrece para trabajar con bases de datos/experimentos remotos: Oracle, MySQL, PostgreSQL, SQLite, HSQL o por ODBC.

Define seis interfaces para el trabajo con tareas remotas, con diferentes tipos de resultados…

5.4.9 Paquete filters

El paquete *filters* contiene todas las clases necesarias para poder trabajar con filtros en *Weka* (desde la pestaña de preprocesado del explorador). Contiene una serie de clases que en su mayoría son clases abstractas y declaración de interfaces, y tiene definidos los paquetes *supervised* y *unsupervised*, que a su vez contienen los paquetes *instance* y *attribute*. En cada uno de estos cuatro subpaquetes se implementan los correspondientes filtros. Por tanto, se establecen los siguientes tipos de filtros:

- Supervisados de instancia (tres filtros)
- Supervisados de atributo (seis filtros)
- No supervisados de instancia (13 filtros)
- No supervisados de atributo (46 filtros)

Fuera de estos subpaquetes, es decir, directamente bajo el paquete *filters*, también declara dos filtros que no tienen cabida en la categorización anterior. Los dos filtros implementados disponibles para el usuario son:

- *AllFilter*: Filtro que no realiza ninguna selección, sino que todas las instancias pasan a través de él sin ser modificadas.

- *MultiFilter*: Filtro que permite seleccionar un conjunto de filtros y que serán aplicados sucesivamente cuando se aplique.

También define tres interfaces que son de utilidad a la hora de implementar los filtros, estas son:

- *StreamableFilter*: Interfaz para filtros que puedan trabajar con un flujo de instancias.
- *SupervisedFilter*: Interfaz para filtros supervisados, es decir, aquellos que hacen uso del atributo clase.
- *UnsupervisedFilter*: Interfaz para filtros no supervisados, es decir, aquellos que no necesitan conocer el valor de la clase.

5.4.10 Paquete gui

El paquete *gui* contiene todas las clases necesarias para poder trabajar con *Weka* desde la interfaz gráfica. Contiene, a su vez, los siguientes paquetes:

- *arffviewer*
- *beans*
- *boundaryvisualizer*
- *ensembleLibraryEditor*
- *experiment*
- *explorer*
- *graphvisualizer*
- *images*
- *sql*
- *streams*
- *treevisualizer*
- *visualize*

Como puede observarse, estos subpaquetes contienen las clases necesarias para visualizar en formato tabular ficheros arff, para mostrar las ventanas de *Experimenter* y *Explorer*, para visualizar árboles y grafos…

Además, el paquete *gui*, contiene ficheros de configuración, algunos de los cuales son:

- *GenericObjectEditor.props*: Lista de todos los estimadores, clasificadores, filtros… que el usuario tendrá disponibles para utilizar. Si se desarrolla un nuevo algoritmo, para que aparezca y pueda ser utilizado desde la interfaz gráfica ha de añadirse en la lista correspondiente.
- *GenericPropertiesCreator.props*: Igual al anterior, pero no llega a nivel de clase. Es utilizado para mostrar al usuario en directorios los clasificadores, filtros… de que dispone.
- *LookAndFeel.props*: Para configurar la apariencia de la interfaz.
- *MemoryUsage.props*: Para configurar el panel de uso de memoria.

- *SimpleCLI.props*: Dispone de una variable para indicarle el máximo número de comandos anteriores a ser recordados.

El paquete *gui* define tres interfaces:

- *CustomPanelSupplier*: Interfaz para objetos capaces de proporcionar su propios componentes para la GUI.
- *Logger*: Interfaz para objetos que muestran *logs* y mensajes de estado.
- *TaskLogger*: Interfaz para objetos que muestran *logs* y demás información durante la ejecución de tareas.

5.5 Desarrollo de algoritmos en Weka

En este apartado se explicará cómo añadir un nuevo algoritmo que, a modo de ejemplo, será un meta-algoritmo que llamaremos *NuevoAlgoritmo*. Su funcionamiento será muy simple, en realidad, no realizará ninguna tarea sobre los resultados que el clasificador base le proporcione, sino que simplemente se limitará a invocarlo.

En primer lugar, debe crearse la clase *NuevoAlgoritmo.java* en el paquete correspondiente. En este caso, la nueva clase deberá crearse en el paquete *weka.classifiers.meta*. Dicha clase deberá heredar obligatoriamente de *Classifier.java* o de alguna de las clases que, a su vez, heredan de *Classifier.java* y amplían sus características, por ejemplo, la clase *RandomizableIteratedSingleClassifierEnhancer*.

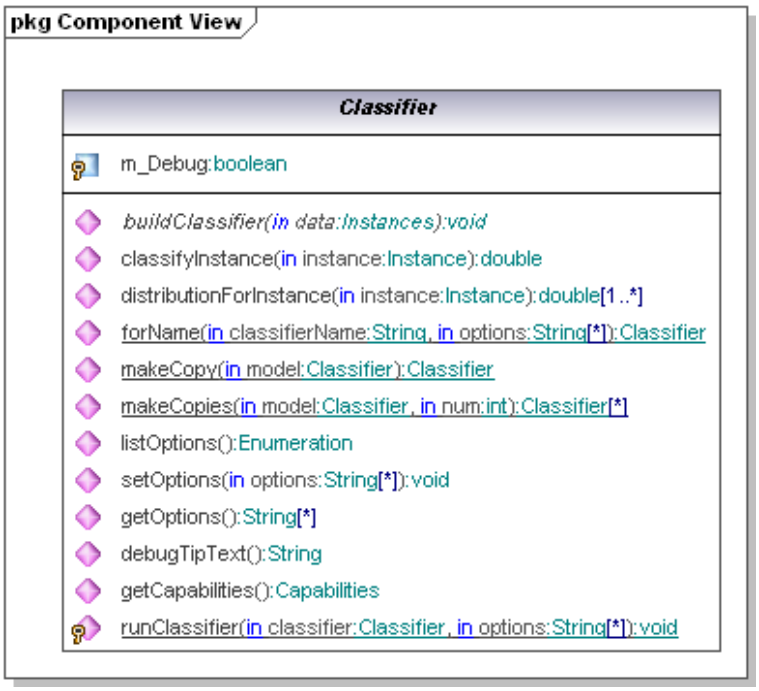

Ilustración 5.18 – Diagrama de clases de *Classifier.java*

Antes de comenzar con la implementación del algoritmo es necesario entender *Classifier.java*. En la ilustración pueden observarse sus atributos y métodos más importantes. Su único atributo es *m_Debug*, un booleano que indica si el usuario ha activado la depuración y desea que se le suministre más información. Sus métodos son los siguientes:

- *void buildClassifier(Instances data)*: Este método es invocado por *Weka*[6]. Como parámetros se recibe el conjunto de instancias de entrenamiento, por tanto, podrá ser invocado más de una vez por ejecución dependiendo del modo de test elegido.

[6] Concretamente, desde el método *startClassifier()* de la clase *ClassifierPanel.java*, que se encuentra en el paquete *weka.gui.explorer*.

En cada llamada debe construirse el clasificador con las instancias enviadas como parámetro, sin que estas sean modificadas. Además, si el clasificador cuenta con opciones (atributos), cuyos valores no pueden ser establecidos desde la ventana de opciones del clasificador, este método es el lugar adecuado para inicializarlas.

- *double classifyInstance(Instance instance)*: Este método es invocado tras la construcción del clasificador, es decir, tras la invocación del método *buildClassifier*. Recibe como parámetros una instancia de test (cuyo atributo clase no es nominal), por tanto, será invocado tantas veces como instancias tenga el conjunto de test (recuerda que dependiendo del modo de test elegido puede haber más de un conjunto de test).
  Este método devuelve un valor real que se corresponde con la predicción que hace para la instancia dada.
  Un clasificador debe implementar este método o *distributionForInstance* (o ambos).
- *double[] distributionForInstance(Instance instance)*: Este método es invocado tras la construcción del clasificador, es decir, tras la invocación del método *buildClassifier*. Recibe como parámetros una instancia de test (cuyo atributo clase es nominal), por tanto, será invocado tantas veces como instancias tenga el conjunto de test (recuerda que dependiendo del modo de test elegido puede haber más de un conjunto de test).
  Este método devuelve un valor real para cada posible valor de la clase, que indican las probabilidades de que la instancia dada pertenezca a dicha clase.
  Un clasificador debe implementar este método o *classifyInstance* (o ambos).
- *Classifier forName(String classifierName, String[] options)*: Crea una nueva instancia del clasificador con el nombre proporcionado como parámetro y opciones para su método *setOptions()*.
- *Classifier makeCopy(Classifier model)*: Crea una copia del clasificador usando serialización.
- *Classifier[] makeCopies(Classifier model, int num)*: Crea tantas copias del clasificador (usando serialización) como se le indique como segundo argumento.
- *Enumeration listOptions()*: Lista las opciones que tiene el clasificador. Este método se invoca desde consola cuando se intenta ejecutar el clasificador sin indicarle el valor de todas las opciones.
- *void setOptions(String[] options)*: Carga el valor que el usuario ha introducido para las opciones desde la ventana de diálogo del clasificador.
- *String[] getOptions()*: Enumera las opciones seleccionadas por el usuario y las devuelve contenidas en un array de cadenas. Este método es invocado justo tras empezar la ejecución del clasificador y es mostrado por *Weka* en la primera línea de la pantalla de salida antepuesto al nombre del clasificador.

A modo de ejemplo, para el clasificador J48 aparece destacada la información devuelta por este método:

```
=== Run information ===

Scheme:       weka.classifiers.trees.J48 -C 0.25 -M 2
Relation:     pima_diabetes
…
```

- *String debugTipText()*: Devuelve un String que será mostrado junto a la opción de depuración en la ventana de diálogo del clasificador. Por tanto, se explicaría en qué consiste dicha opción, que valores puede tomar…
- *Capabilities getCapabilities()*: Devuelve las capacidades/habilidades del clasificador. Por ejemplo si puede trabajar con clases numéricas, atributos nominales…
- *void runClassifier(Classifier classifier, String[] options)*: Ejecuta la instancia de clasificador proporcionada como primer argumento con las opciones indicadas como segundo parámetro.

Una vez entendidas las clases *Classifiers.java*, *Attribute.java*, *Instance.java* e *Intances.java*, se dispone de los conocimientos suficientes para comenzar el desarrollo de nuestro sencillo meta-algoritmo.

Como se dijo previamente, se debe crear la clase *NuevoAlgoritmo.java* en el paquete *weka.classifiers.meta*. En vez de heredar de *Classifier.java*, heredará de *SingleClassifierEnhancer.java* que hereda, a su vez, de *Classifier.java*.

SingleClassifierEnhancer.java nos proporciona, además del atributo *m_Debug*, el atributo *m_Classifier*, es decir, el clasificador base que se usará. Además, lo muestra como opción para que el usuario seleccione el que desee y se actualiza con el valor seleccionado por el usuario. Por tanto, puesto que todos los meta-algoritmos necesitan, al menos, un clasificador base, esta clase abstracta es de gran ayuda en el desarrollo de los mismos.

A continuación se muestra el código del meta-algoritmo:

```
package weka.classifiers.meta;

import weka.classifiers.SingleClassifierEnhancer;
import weka.core.Instance;
import weka.core.Instances;

public class NuevoAlgoritmo extends SingleClassifierEnhancer{

	private static final long serialVersionUID = 3738977858533409501L;

	public NuevoAlgoritmo() {
		m_Classifier = new weka.classifiers.trees.J48();
```

```
	}

	public void buildClassifier(Instances data) throws Exception {
		m_Classifier.buildClassifier(data);
	}

	public String getRevision() {
		return null;
	}

	public double classifyInstance(Instance instance) throws Exception {
		return m_Classifier.classifyInstance(instance);
	}

	public static void main(String [] argv) {
		runClassifier(new NuevoAlgoritmo(), argv);
	}
}
```

Como puede observarse, el código necesario es minúsculo, puesto que los métodos *buildClassifier* y *classifyInstance* sólo tienen que llamar a los métodos del clasificador base. Además, puesto que es obligatorio, también se implementa *getRevision()*.

El método *main* sólo es necesario si se desea que sea ejecutable desde la consola (*SimpleCLI*). Desde este método se llama el método *runClassifier*, implementado en *Classifier.java*, con las opciones recibidas. Además, se aprovecha la obligada construcción del constructor (puesto que se ha implementado el método *main*) para cambiar el algoritmo base por uno más habitual: *J48*.

Por último, para que el meta-algoritmo esté disponible para su uso debe indicársele a *Weka* que lo muestre por la interfaz gráfica. Para ello ha de añadirse en el fichero *GenericObjectEditor.props*, que se encuentra en el paquete *weka.gui*. En este fichero se indican las clasificadores existentes, filtros… El contenido del fichero quedaría como se muestra a continuación:

```
...
# Lists the Tokenizers I want to choose from
weka.core.tokenizers.Tokenizer=\
 weka.core.tokenizers.AlphabeticTokenizer,\
 weka.core.tokenizers.NGramTokenizer,\
 weka.core.tokenizers.WordTokenizer

# Lists the Classifiers I want to choose from
weka.classifiers.Classifier=\
 weka.classifiers.bayes.AODE,\
 ...
 weka.classifiers.meta.MultiScheme,\
 weka.classifiers.meta.NuevoAlgoritmo,\
 weka.classifiers.meta.OrdinalClassClassifier,\
 ...
 weka.classifiers.rules.Ridor,\
 weka.classifiers.rules.ZeroR

# Lists the DistanceFunctions I want to choose from
weka.core.DistanceFunction=\
 weka.core.ChebyshevDistance,\
```

```
weka.core.EditDistance,\
...
```

Una vez finalizado el desarrollo, compilado y ejecutado, puede verse que *NuevoAlgoritmo* aparece disponible entre los meta-algoritmos y al seleccionarlo ofrece dos opciones en su ventana de diálogo: depuración y clasificador base, ambas heredadas.

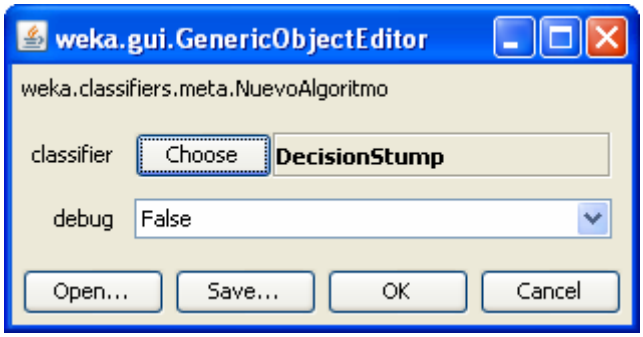

Ilustración 5.19 – Ventana de opciones de *NuevoAlgoritmo*

Ejecutándose se observa que tiene el mismo funcionamiento que el clasificador base que se tenga seleccionado en ese momento.

6. Descripción del Trabajo Realizado

En este capítulo se describirá qué desarrollo se ha llevado a cabo en este proyecto. En primer lugar, se indicará qué se pretende conseguir y con qué herramientas/tecnologías va a ser desarrollado. Posteriormente, se indicará la integración en Weka y su implementación a alto nivel. En último lugar se explicarán qué ficheros de salida genera el algoritmo desarrollado y se ilustrará un ejemplo de ejecución.

6.1 Introducción

En este apartado se resumirá qué se pretende conseguir con este proyecto, se listarán en forma de requisitos de usuario las características necesarias/requeridas/deseadas del algoritmo a desarrollar y se indicará qué herramientas y tecnologías se emplearán en su construcción.

6.1.1 Objetivos

El algoritmo a desarrollar tiene como objetivo construir un procedimiento para transformar o proyectar conjuntos de datos (incrementando, manteniendo o disminuyendo su dimensionalidad) de manera que maximice los resultados de clasificación con un algoritmo determinado o de una manera más general. La idea central de las transformaciones y proyecciones es cambiar el espacio en el que se representan los datos para mejorar los resultados de los algoritmos de clasificación.

Una forma para mejorar los resultados de los diferentes algoritmos y obtener algo equivalente a un espacio proyectado es emplear diferentes funciones de distancia. En la ilustración 6.1 puede verse la ecuación de la distancia euclídea clásica:

$$d_E(\mathbf{x},\mathbf{c}) = \sqrt{\sum_{i=1}^{n}(x_i - c_i)^2} = \sqrt{(\mathbf{x}-\mathbf{c})^T(\mathbf{x}-\mathbf{c})}$$

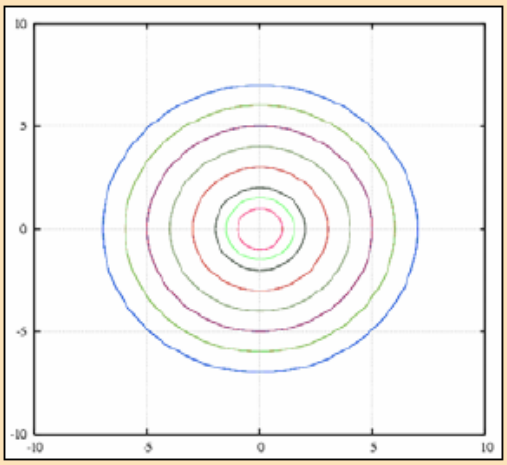

Ilustración 6.1 – Fórmula de la distancia euclídea clásica

Las circunferencias representar puntos situados a la misma distancia, por tanto, la distancia euclídea clásica mide la distancia real entre los puntos.

En las ilustraciones 6.2 y 6.3 pueden verse las ecuaciones de la distancia euclídea ponderada y generalizada respectivamente.

$$d_d(\mathbf{x},\mathbf{c}) = \sqrt{\sum_{i=1}^{n}(m_i(x_i - c_i))^2} = \sqrt{(\mathbf{x}-\mathbf{c})^T M^T M(\mathbf{x}-\mathbf{c})}$$

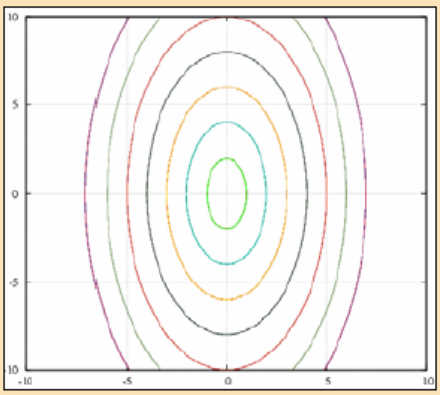

Ilustración 6.2 – Fórmula de la distancia euclídea ponderada

$$d_M(\mathbf{x},\mathbf{c}) = \sqrt{(\mathbf{x}-\mathbf{c})^T M^T M(\mathbf{x}-\mathbf{c})}$$

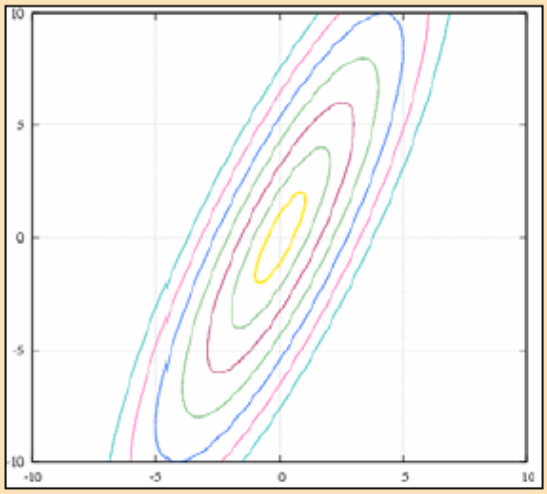

Ilustración 6.3 – Fórmula de la distancia euclídea generalizada

Puede observarse que, en el caso de la distancia euclídea ponderada y generalizada, las distancias dependen de una matriz M, por lo que utilizando distintas matrices m tendremos distintas funciones de distancia. Puesto que el problema de la clasificación consiste en el fondo en separar las regiones del espacio que pertenezcan a distintas clases, puede ser adecuado el redefinir las relaciones de cercanía entre los datos para mejorar la separabilidad de los datos. Es decir, puede ser conveniente acercar ciertos datos y alejar otros, y esto se puede conseguir modificando la matriz M.

En la ilustración 6.3 puede verse cómo actúa la distancia euclídea generalizada. Las elipses representan puntos situados a la misma distancia, por lo que puntos que antes estaban lejos del centro (situados en la parte excéntrica de la elipse), al usar dicha distancia se acercan.

En el presente trabajo, en lugar de funciones de distancia, se utilizarán proyecciones para transformar directamente el espacio en el que se representan los datos. Dichas transformaciones también se representan con matrices, las cuales serán optimizadas mediante técnicas evolutivas. Se utilizan transformaciones porque estas se pueden aplicar a cualquier algoritmo de aprendizaje automático, y no sólo aquellos que utilicen distancias.

Las transformaciones que se van a considerar consisten en multiplicar los datos originales por una matriz. Si llamamos **x** al vector que contiene los atributos de entrada de los datos representados en el espacio original, $\mathbf{x'}=(\mathbf{x}\cdot M)^n$ representará los datos transformados. En esta expresión, M es la matriz que realiza la transformación y n es el exponente que determina la linealidad de la transformación. M puede ser:

- Una matriz cuadrada, en cuyo caso las dimensiones de los datos transformados **x'** es la misma que la de los originales **x**.

- Una matriz rectangular de tamaño nxm, donde n son las dimensiones del espacio original y m las del proyectado. Si m<n reduciremos la dimensionalidad y si m>n la aumentaremos.

También se considerarán tres tipos de matrices: diagonales, simétricas y completas.

- Las matrices diagonales sólo tienen valores en la diagonal y ceros en el resto.
- Las matrices simétricas tienen los mismos valores en el triángulo inferior izquierdo que en el triángulo superior derecho.
- Las completas tienen todos sus valores.

Como se puede ver, unas matrices consideran más parámetros (completas) que otras (diagonales). El uso de muchos parámetros puede inducir sobreadaptación, mientras que el uso de pocos puede llevar a subadaptación. La matriz más apropiada dependerá del problema y se determinará mediante experimentación.

En cuanto al exponente n, si n=1, las transformaciones serán lineales, mientras que si n>1 serán no lineales (cuadráticas si n=2…).

El objetivo principal del proyecto es utilizar técnicas evolutivas para encontrar la matriz M que optimice el porcentaje de aciertos en problemas de clasificación (o el error cuadrático en problemas de regresión). Es decir, se espera que para un algoritmo de clasificación dado, que denominaremos algoritmo base (por ejemplo, *J48*), el porcentaje de aciertos en test para el conjunto de datos transformados **x'** sea mas alto que para el conjunto original **x**. La matriz M no es más que un conjunto de valores reales, y se sabe que las técnicas evolutivas son apropiadas para optimizar cromosomas compuestos de reales. La función a optimizar (la función de fitness) será el porcentaje de aciertos obtenido al transformar los datos de entrenamiento con el individuo (una matriz M concreta).

En el presente proyecto no sólo se quieren implementar y probar las ideas anteriores, sino integrarlas dentro del sistema de aprendizaje automático *Weka*, para que la transformación de datos pueda ser utilizada con cualquiera de los algoritmos y filtros proporcionados por dicha herramienta. Sin embargo, de hecho, nuestra proyección evolutiva será implementada como un meta-algoritmo, que use otro algoritmo base, que será el beneficiario de la proyección.

A pesar de que *Weka* permite la integración de nuevos algoritmos, este proceso está lejos de ser trivial, por lo que la inclusión de la técnica evolutiva dentro de *Weka* es una parte importante de este proyecto y ha consumido una buena parte del tiempo del mismo.

Por último, también se llevará a cabo una validación experimental del sistema desarrollado.

6.1.2 Requisitos del sistema

En este apartado se indicarán los requisitos de usuario e inversos que debe cumplir el desarrollo.

Los requisitos de usuario que cumple el algoritmo son los siguientes:

Identificador	RU001
Descripción	Meta-algoritmo que, haciendo uso de computación evolutiva, busque la matriz de pesos que proyecte al conjunto de datos cuyo resultado sea el mejor posible.
Necesidad	Esencial
Prioridad	Alta

Tabla 6.1 – Requisito de usuario RU001

Identificador	RU002
Descripción	El meta-algoritmo desarrollado es integrado en Weka y será seleccionable desde la carpeta *meta* (aquella que contiene todos los meta-algoritmos).
Necesidad	Esencial
Prioridad	Alta

Tabla 6.2 – Requisito de usuario RU002

Identificador	RU003
Descripción	El algoritmo base puede ser seleccionado entre cualesquiera de los implementados en Weka.
Necesidad	Esencial
Prioridad	Alta

Tabla 6.3 – Requisito de usuario RU003

Identificador	RU004
Descripción	*PMatrix* debe funcionar correctamente sin importar el modo de test elegido y sus posibles parámetros.
Necesidad	Esencial
Prioridad	Alta

Tabla 6.4 – Requisito de usuario RU004

Identificador	RU005
Descripción	Capacidad para trabajar con cualquier conjunto de datos, sin importar su números de instancias ni su dimensión; siendo Weka, el sistema operativo y/o el computador los culpables de que esta capacidad se viera limitada.
Necesidad	Deseable
Prioridad	Alta

Tabla 6.5 – Requisito de usuario RU005

Identificador	RU006
Descripción	El algoritmo puede trabajar con dominios con clases nominales o numéricas.
Necesidad	Esencial
Prioridad	Alta

Tabla 6.6 – Requisito de usuario RU006

Identificador	RU007
Descripción	El sistema utiliza como cálculo de la fitness para un individuo (matriz de pesos) el porcentaje de instancias clasificadas correctamente si la clase es nominal o el error cuadrático medio si la clase es numérica.
Necesidad	Esencial
Prioridad	Alta

Tabla 6.7 – Requisito de usuario RU007

Identificador	RU008
Descripción	El meta-algoritmo *PMatrix* genera un directorio por cada ejecución conteniendo un fichero de traza y dos subdirectorios, *arff* y *models*, que contienen los ficheros *arff* de los conjuntos de training y test proyectados y los modelos que construye el algoritmo base para los conjuntos de training y test proyectados.
Necesidad	Opcional
Prioridad	Baja

Tabla 6.8 – Requisito de usuario RU008

Identificador	RU009
Descripción	El usuario puede activar o desactivar la depuración. En caso de

	estar activada, *PMatrix* genera dos ficheros de texto adicionales: *debug* y *debugP*, conteniendo depuración del proceso de búsqueda y de las proyecciones de conjuntos de datos e instancias, respectivamente.
Necesidad	Deseable
Prioridad	Baja

Tabla 6.9 – Requisito de usuario RU009

Identificador	RU010
Descripción	El meta-algoritmo puede trabajar, a elección del usuario, con matrices completas, simétricas o diagonales.
Necesidad	Esencial
Prioridad	Media

Tabla 6.10 – Requisito de usuario RU010

Identificador	RU011
Descripción	El usuario puede elegir la dimensionalidad del espacio proyectado. Por tanto, puede aumentarla, disminuirla o mantenerla.
Necesidad	Deseable
Prioridad	Media

Tabla 6.11 – Requisito de usuario RU011

Identificador	RU012
Descripción	El usuario puede elegir el exponente a aplicar tras la proyección con la matriz.
Necesidad	Opcional
Prioridad	Media

Tabla 6.12 – Requisito de usuario RU012

Identificador	RU013
Descripción	La población seleccionada para el cruce es elegida con el método de los torneos, cuyo tamaño puede elegir el usuario.
Necesidad	Deseable
Prioridad	Alta

Tabla 6.13 – Requisito de usuario RU013

Identificador	RU014
Descripción	El meta-algoritmo tiene otros parámetros configurables por el usuario, estos son: constante de decremento, número de nuevos individuos generados en el cruce, número de generaciones sin mejora que continuará la búsqueda, tamaño de la población, probabilidad de mutación de un individuo y probabilidad de mutación de cada una de las entradas de su matriz de pesos.
Necesidad	Deseable
Prioridad	Media

Tabla 6.14 – Requisito de usuario RU014

Identificador	RU015
Descripción	El idioma de las opciones mostradas por el algoritmo por la interfaz de Weka será, al igual que el resto de algoritmos, el inglés.
Necesidad	Esencial
Prioridad	Alta

Tabla 6.15 – Requisito de usuario RU015

Identificador	RU016
Descripción	El sistema asigna unos valores iniciales a las variables configurables, de tal forma que no es obligatorio que el usuario elija uno para ellas.
Necesidad	Esencial
Prioridad	Alta

Tabla 6.16 – Requisito de usuario RU016

Los requisitos inversos son los siguientes:

Identificador	RI001
Descripción	El meta-algoritmo no podrá trabajar con conjuntos de datos con los que no pueda trabajar el algoritmo base seleccionado.

Tabla 6.17 – Requisito inverso RI001

Identificador	RI002
Descripción	La nueva versión ejecutable de Weka con *PMatrix* integrado debe funcionar obligatoriamente en Windows, no teniendo porqué funcionar en el resto de plataformas/sistemas operativos.

Tabla 6.18 – Requisito inverso RI002

6.1.3 Desarrollo del sistema

El algoritmo ha sido desarrollado utilizando el lenguaje JAVA, necesario para la integración en Weka, en concreto, *PMatrix* ha sido integrado en la versión de desarrollador 3.5.8.

Durante el desarrollo del algoritmo ha sido utilizado el entorno de desarrollo de software libre *Eclipse*, en su versión 3.3.0, configurado con *JDK* 1.6.

La compilación con *Eclipse* se realizará con el uso de *Ant*, de manera que se genere un fichero con extensión *jar* con que puede ejecutarse y probar directamente el algoritmo desarrollado.

Los diagramas de clases han sido creados con la herramienta *Altova UModel 2008 © Enterprise Edition*.

A modo de ejemplo, para comprender los diagramas incluidos en este capítulo se muestra la siguiente ilustración, donde se pueden observar las peculiaridades con que el software empleado muestra los diferentes tipos de variables, métodos y clases:

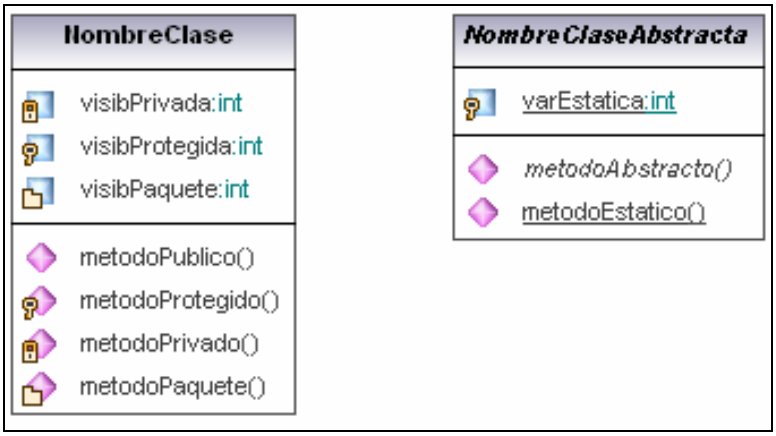

Ilustración 6.4 – Diagrama de clases de ejemplo

6.2 Integración en Weka

En este apartado se expondrán las dificultades que conlleva la integración de un algoritmo de estas características en Weka. El algoritmo ha de ser añadido en Weka sin modificar absolutamente nada en el resto de clases.

6.2.1 Flujo de datos en Weka

Como ya se ha comentado, en *Weka* existen cuatro modos de test, que se explicarán de un modo más orientado al flujo de datos y al desarrollo:

- **Use training set**: *Weka* invoca al método *buildClassifier*() de nuestro algoritmo con el conjunto de datos original. Posteriormente, invoca a los métodos *classifyInstance*() o *distributionForInstance*() tantas veces como instancias tenga el conjunto de datos original. El método *buildClassifier*() habrá sido invocado una vez durante la ejecución.
- **Supplied test set**: *Weka* invoca al método *buildClassifier*() de nuestro algoritmo con el conjunto de datos original. Posteriormente, invoca a los métodos *classifyInstance*() o *distributionForInstance*() tantas veces como instancias tenga el conjunto de datos proporcionado adicionalmente. El método *buildClassifier*() habrá sido invocado una vez durante la ejecución.
- **Cross-validation**: *Weka* invoca al método *buildClassifier*() de nuestro algoritmo con el conjunto de datos original. Posteriormente, entra en un bucle con tantas iteraciones como hojas haya seleccionado el usuario. En cada iteración invoca al método *buildClassifier*() con el conjunto de training "temporal" y a los métodos *classifyInstance*() o *distributionForInstance*() con el conjunto de test "temporal". El método *buildClassifier*() habrá sido invocado *<<número_de_hojas+1>>* veces durante la ejecución.
- **Percentage split**: *Weka* invoca al método *buildClassifier*() de nuestro algoritmo con el conjunto de datos original, de nuevo con el conjunto de datos reducido al porcentaje indicado y, por último, invoca a los métodos *classifyInstance*() o *distributionForInstance*() con el porcentaje de instancias restantes. El método *buildClassifier*() habrá sido invocado dos veces durante la ejecución.

Weka representa estos cuatro modos con una variable entera, cuyo valor oscila entre uno y cuatro[1]. Además, *Weka* almacena el número de hojas seleccionado si se ha elegido validación cruzada, el porcentaje deseado si se ha elegido *Percentage split* y el conjunto de datos de test si se ha elegido *Supplied test set*. Por todo esto, desde el algoritmo es imposible conocer cuántas veces será invocado nuestro método *buildClassifier*().

[1] Valor uno para *Cross-validation*; valor dos para *Percentage split*; valor tres para *Use training set* y valor cuatro para *Supplied test set*.

6.2.2 Peculiaridades de implementación

Un clasificador en Weka, básicamente, está supuesto a hacer lo siguiente:

1. Construir un modelo, más o menos complejo, con los datos proporcionados como argumento en la llamada a *buildClassifier*().
2. Evaluar con el último modelo construido las instancias que Weka proporcione como argumento en los métodos *classifyInstance*() o *distributionForInstance*().
3. Si *Weka* vuelve a invocar el método *buildClassifier*(), se reemplaza el modelo construido por el nuevo modelo que se construya con los nuevos datos que haya proporcionado.

Sin embargo, el algoritmo *PMatrix* no podría ser desarrollado siguiendo esa filosofía, ya que la generación de los ficheros de salida obliga a conocer qué modo de test (y con qué parámetros) ha elegido el usuario. Sin embargo, estos valores no pueden ser conocidos por el clasificador, por tanto, el desarrollo se complica.

Una solución sería modificar el núcleo de Weka para adaptarlo a nuestras necesidades, sin embargo, no es responsable actuar de este modo. La decisión tomada ha sido la siguiente:

1. Se obliga a que el usuario siempre active la opción *Output Model*, que implica que se invoque a *buildClassifier*() con el conjunto de datos cargado por completo. Este será almacenado en la variable *original*.
2. Se necesita conocer cuál es la primera vez que se invoca a mi método *buildClassifier*() para almacenar el conjunto de datos original. Esta información la indicará la variable booleana *primeraVez*.
3. Inicializo la variable *primeraVez* gracias al método *setOptions*(), que es llamado al comienzo de la ejecución.
4. Genero el modelo para la mejor proyección del conjunto de datos original.
5. En las posibles sucesivas llamadas a *buildClassifier*()[2] que Weka puede hacer dependiendo del modo de test elegido, construyo el modelo (previa búsqueda de la mejor matriz de pesos) con el conjunto de datos proyectado y genero los ficheros de salida para el conjunto de datos de entrenamiento actual. Además, el conjunto de datos de test correspondiente lo calculo haciendo la diferencia entre el conjunto de datos original y el actual.
6. Cada vez que me llamen a los métodos *classifyInstance*() y *distributionForInstance*(), me limito (al igual que para construir el modelo) a indicar al algoritmo base que clasifique dicha instancia previa proyección con la última mejor matriz de pesos.

[2] Desde el algoritmo se desconoce si el método *buildClassifier*() volverá a ser invocado puesto que se desconoce el modo de test elegido por el usuario.

Esto funciona independientemente de los modos de test elegido gracias al orden en que Weka llama a los métodos *buildClassifier*(), *classifyInstance*() y *distributionForInstance*(), como se indicó en el apartado 6.2.1.

Un problema de que adolece esta solución es que el método *setOptions*() no sólo es invocado al comienzo de la ejecución del algoritmo, sino también cuando la ventana de la interfaz de Weka pierde y recupera el foco (*window focus*). Por tanto, cada vez que esto ocurra la variable *primeraVez* volverá a ser inicializada sin haber sido la *primeraVez* que se llama al método *buildClassifier*(), por lo que la generación de ficheros de salida provocará un error al acceder a archivos y/o directorios no existentes.

Otro problema importante al que ha de enfrentarse en el desarrollo del algoritmo es la poca funcionalidad que ofrece Weka para trabajar con conjuntos de datos, instancias y atributos. Algo tan simple como establecer un atributo como clase en un conjunto de datos o modificar el nombre de un atributo no pueden llevarse a cabo. En su lugar, han de tomarse soluciones más enrevesadas que produzcan los mismos resultados.

El mejor ejemplo de lo anterior puede observarse en la función *adaptar*() de la clase *PMatrix*, en el que para construir la estructura del conjunto de datos proyectado, ha de partirse del conjunto de datos con que Weka invoca a tu método, eliminando todos sus atributos excepto el clase y creando los atributos a añadir para poder darles un nombre diferente.

6.3 Diagrama de clases

En este apartado se explicarán qué clases se han creado para el desarrollo del algoritmo *PMatrix*. Además, se mostrará el diagrama de clases completo, explicando las relaciones que se dan entre cada una de las clases.

El sistema está compuesto por las siguientes clases:

- **Individuo**: Representa cada uno de los individuos de la población del algoritmo evolutivo.
- **Auxiliar**: Clase que, como su propio nombre indica, contiene métodos estáticos implementados proporcionando una funcionalidad que Weka no otorga. Por ejemplo: la diferencia de conjuntos de datos, proyecciones de instancias y conjuntos de datos…
- **MotorGen**: Clase que, al igual que Auxiliar, contiene numerosos métodos estáticos que permiten realizar la búsqueda de la mejor matriz con computación evolutiva. Contiene métodos para crear poblaciones, mutarlas,
- **PMatrix**: Clase que implementa el núcleo del algoritmo. Contiene métodos obligatorios (para construir el clasificador y clasificar instancias de test, para proporcionar y recoger información por la interfaz) y otros adicionales para ejecutar el algoritmo evolutivo… Es la clase encargada de generar la mayor parte de los ficheros de salida y, para ofrecer dicha funcionalidad, hace uso del resto de clases

El diagrama de clases es un diagrama estático que indica las relaciones que se dan entre cada una de las clases (los atributos y métodos de cada clase serán detallados en los sucesivos apartados):

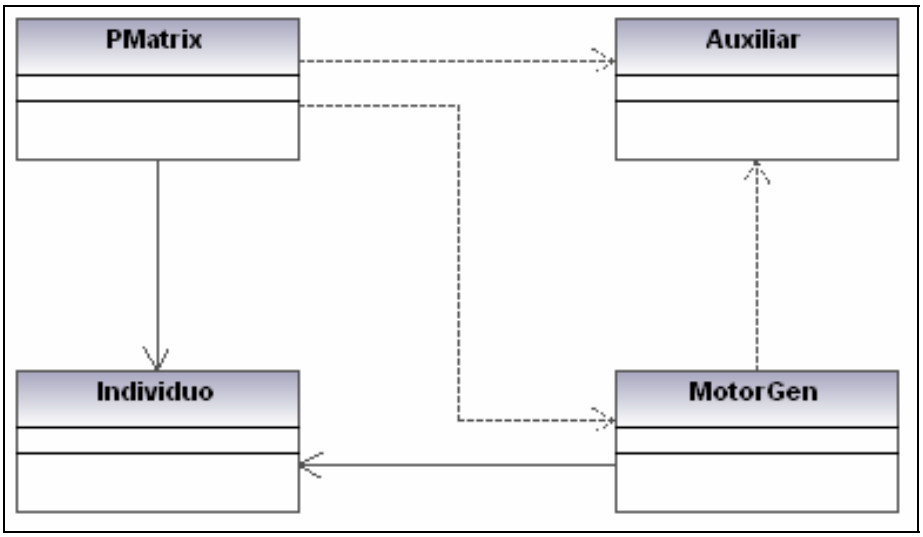

Ilustración 6.5 – Diagrama de clases del meta-algoritmo

Su integración en Weka exige que dichas clases pertenezcan a un paquete u otro dependiendo de si es un algoritmo o clases auxiliares al algoritmo.

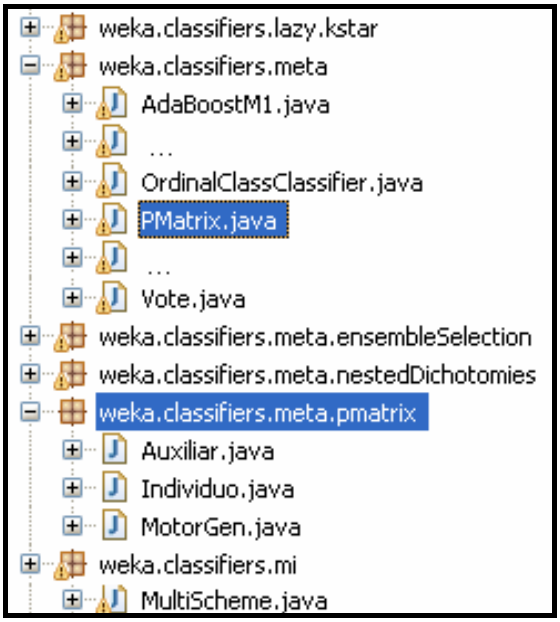

Ilustración 6.6 – Estructura de paquetes en *Weka*

Como puede observarse, el paquete *weka.classifiers.meta* contiene los subpaquetes *ensembleSelection* y *nestedDichotomies* que contienen clases auxiliares para los meta-algoritmos *EnsembleSelection* y *END* respectivamente, perteneciendo las clases principales de dichos meta-algoritmos, *EnsembleSelection.java* y *END.java*, al paquete *weka.classifiers.meta*. Por este motivo, las clases Auxiliar, MotorGen e Individuo han sido creadas en el subpaquete *pmatrix*, del siguiente modo:

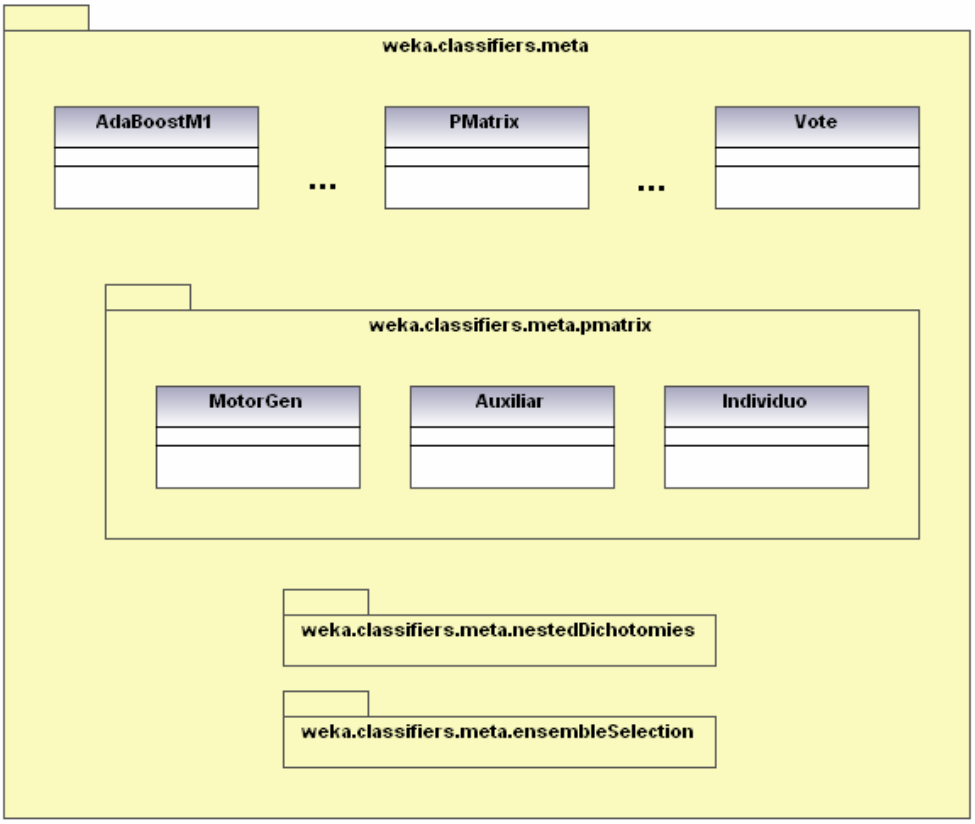

Ilustración 6.7 – Estructura de paquetes desarrollada para *PMatrix*

En los siguientes apartados se explicará detalladamente cada una de las clases creadas para el desarrollo de *PMatrix*. En estos apartados se explicarán decisiones de implementación y detalles de implementación a alto nivel. Para detalles de implementación a bajo nivel ver *Anexo A*.

6.3.1 Clase Individuo

La clase Individuo representa cada uno de los individuos que componen las poblaciones que se emplearán en la búsqueda de la matriz de pesos que mejor proyecte el conjunto de datos.

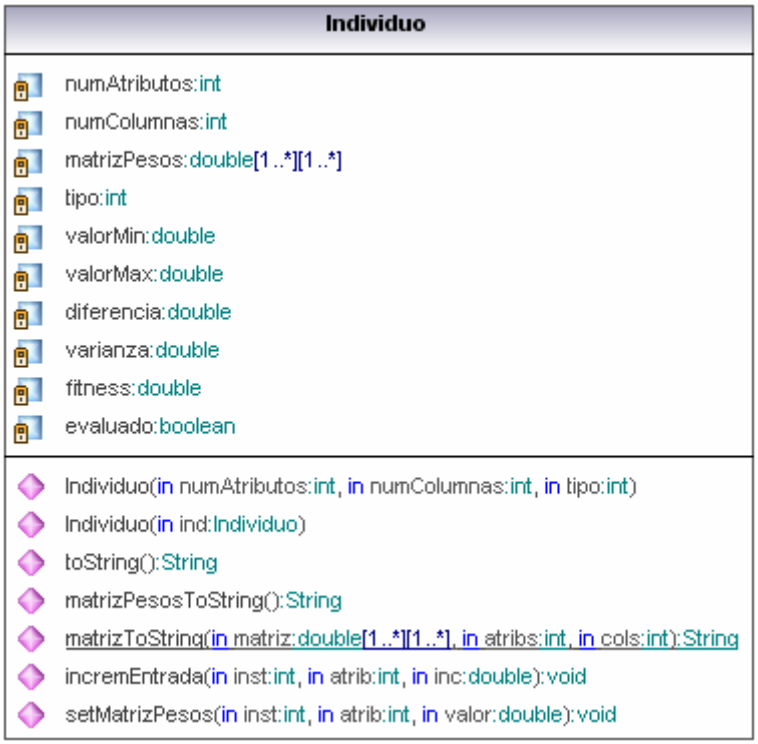

Ilustración 6.8 – Clase *Individuo*

Como puede observarse, un individuo no es más que una matriz de pesos con unas determinadas dimensiones (números de atributos del conjunto de datos x número de columnas seleccionadas por el usuario[3]) cuyos valores iniciales varían en el rango [*valorMin, valorMax*].

Una vez ha comenzado la búsqueda los valores que pueden tomar las entradas de la matriz no están limitados, pudiendo tomar cualquier valor en el dominio de los números reales[4].

La variable *tipo* indica si el individuo es una matriz completa, una matriz simétrica o una matriz diagonal. El valor de esta variable influye en el constructor no parametrizado y en el método *incremEntrada*(), que incrementará una determinada entrada si su valor puede ser distinto de nulo (si individuo es matriz diagonal) y, en caso de ser simétrica, incrementará también la entrada simétrica.

6.3.2 Clase Auxiliar

La clase Auxiliar es una clase cuyos atributos y métodos son todos estáticos. Esta clase es similar, en estructura, a la clase *Math* de Java. Ambas tienen en común que sus métodos pueden ser invocados sin crear un objeto de la clase correspondiente

[3] El número de columnas establecido por el usuario determina el número de atributos (dimensiones) en el espacio destino (proyectado).
[4] Sin tener en cuenta las limitaciones de representación de números en un computador.

ofreciendo una funcionalidad adicional, en el caso de Auxiliar, para el trabajo con conjunto de datos, instancias…

Ilustración 6.9 – Clase *Auxiliar*

La clase Auxiliar ofrece las siguientes funcionalidades a la clase PMatrix[5]:

- El método *diferencia*() calcula la diferencia entre dos conjuntos de instancias y devuelve un dataset con las instancias correspondientes. Es necesario para poder generar como salida el conjunto de datos con las instancias que se utilizarán para test.
- El método *iguales*() comprueba si dos instancias son la misma. Útil para llevar a cabo su funcionalidad el método *diferencia*().
- El método *generarARFF*() genera un archivo de salida con el conjunto de instancias indicado en formato ARFF. El archivo de salida generado podría ser cargado por Weka si se desean visualizar los datos proyectados.
- El método *proyectarDS*() proyecta un conjunto de datos con la matriz de pesos indicada. Este método es invocado en multitud de ocasiones durante la búsqueda de la mejor matriz de pesos.
- El método *proyectarInstancia*() proyecta una instancia de datos con la matriz de pesos indicada. A diferencia del método *proyectarDS*(), este método es utilizado para proyectar cada una de las instancias de test con que Weka invoca como argumento tus métodos *classifyInstance*() y *distributionForInstance*().

Las variables estáticas han sido definidas puesto que son necesarias para la mayor parte de métodos y, de este modo, se evitan pasar como argumento en cada uno de ellos, haciendo la definición de los métodos más simple e intuitiva.

[5] El método *proyectarDS*() es el único que no es invocado siempre desde la clase *PMatrix*, ya que el método *evaluarPoblacion*() de MotorGen hace uso del mismo porque necesita evaluar el dataset proyectado, no el original.

6.3.3 Clase MotorGen

La clase *MotorGen*, al igual que se comentó para la clase *Auxiliar*, es una clase cuyos atributos y métodos son todos estáticos. Como también se dijo, esta clase es similar, en estructura, a la clase *Math* de Java. Ambas tienen en común que sus métodos pueden ser invocados sin crear un objeto de la clase correspondiente ofreciendo una funcionalidad adicional, en el caso de *MotorGen*, para el trabajo con algoritmos evolutivos.

Ilustración 6.10 – Clase *MotorGen*

La clase Auxiliar ofrece las siguientes funcionalidades a la clase *PMatrix*:

- El método *mejora*() indica si se acaba de mejorar el mejor fitness hallado hasta el momento.
- El método *torneo*() realiza una selección de individuos de la población para su posterior cruce realizando el método de torneos.
- El método *cruce*() cruza pares de individuos de la población seleccionada y los muta. Posteriormente, estos nuevos individuos son insertados en la población reemplazando al peor que haya en ese momento[6].
- El método *mutarPoblacion*() realiza una mutación a un individuo con una determinada probabilidad, que consiste en mutar cada una de sus entradas con otra determinada probabilidad.
- El método *calcularAciertos*() proyecta el conjunto de datos con la matriz indicada y evalúa el conjunto de datos proyectado con validación cruzada de 10 hojas. Como medida de aptitud devuelve el porcentaje de instancias clasificadas

[6] Es posible que el peor individuo de la población sea uno de los nuevos individuos que acababa de ser insertado en la población.

correctamente si la clase es nominal y el error cuadrático medio en caso de que sea numérica.

- Los métodos *peor*() y *mejor*() determinan cuál es el peor o el mejor individuo de una población.
- El método *iniciarPoblacion*() crea y evalúa un conjunto de individuos para formar la población inicial.
- El método *evaluarPoblacion*() evalúa a todos los individuos de la población.
- El método *mostrarPoblacion*() escribe en el fichero de depuración (*debug.txt*) todos los individuos de la población con mayor o menor grado de detalle.

Al igual que se comentó para la clase *Auxiliar*, las variables estáticas han sido definidas puesto que son necesarias para la mayor parte de métodos y, de este modo, se evitan pasar como argumento en cada uno de ellos, haciendo la definición de los métodos más simple e intuitiva.

6.3.4 Clase PMatrix

La clase *PMatrix* es la clase que hereda de *Classifier.java*. Como ya se ha comentado, todos los algoritmos implementados en Weka deben heredar, directa o indirectamente, de ella. Por todo esto, *PMatrix* es la clase que implementa el núcleo del meta-algoritmo.

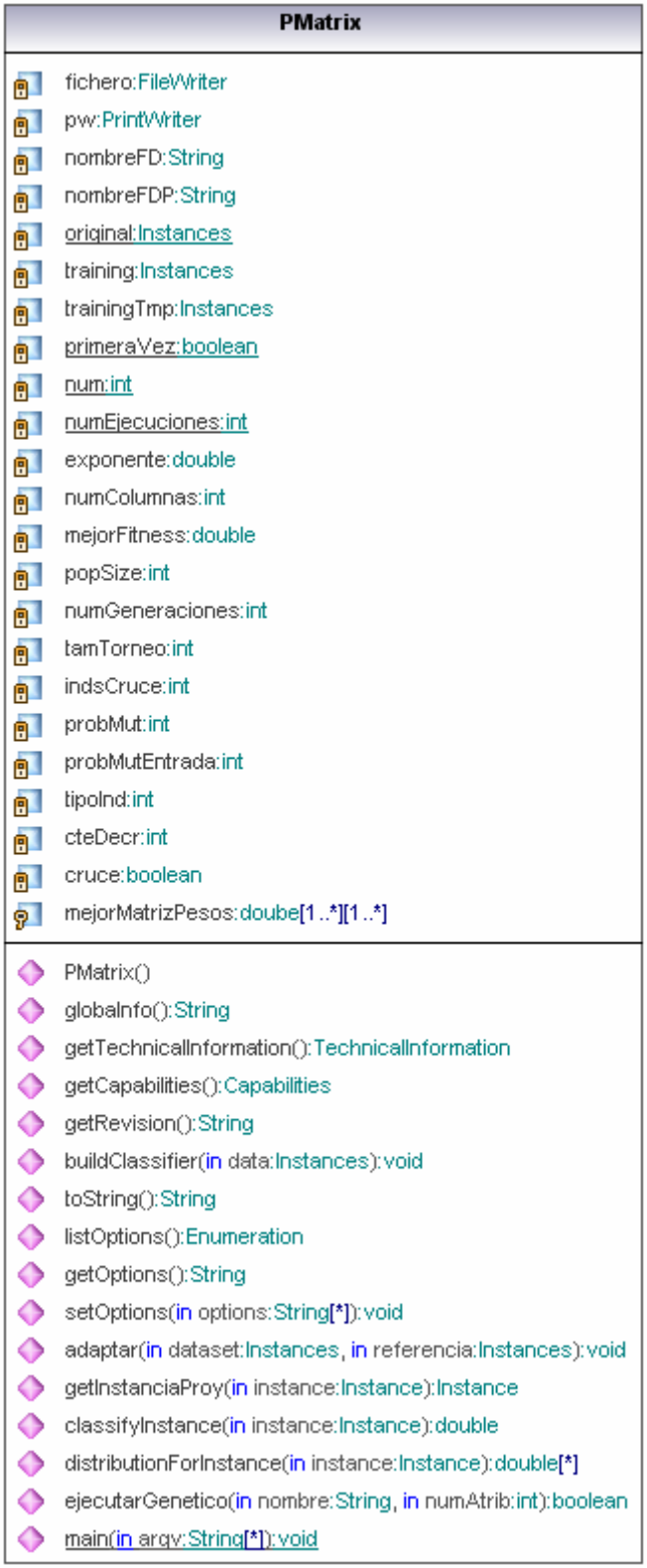

Ilustración 6.11 – Clase *PMatrix*

La clase *PMatrix* define todas las variables cuyos valores pueden ser establecidos desde la ventana de diálogo del meta-algoritmo en la interfaz. Además, se definen variables para facilitar la generación de los ficheros de salida (*fichero*, *pw*, *nombreFD* y *nombreFDP*).

Por último, según se ha explicado en el apartado 6.2, para la correcta generación de los ficheros de salida debe conocerse cuál es la primera vez que desde Weka se invoca el método *buildClassifier*() para construir el clasificador; también ha de saberse cuántas veces ha sido invocado y cuántas veces ha sido ejecutado el meta-algoritmo por el usuario. Por todo esto, son necesarias las variables *primeraVez*, *num* y *numEjecuciones*.

Además, la clase *PMatrix* implementa los siguientes métodos:

- El método *main*() sirve para la ejecución del algoritmo desde la consola de *Weka* (*SimpleCLI*).
- Los métodos *getTechnicalInformation*() y *globalInfo*() devuelven texto que es mostrado en la interfaz por la ventana de diálogo del meta-algoritmo.
- El método *getCapabilities*() indica con qué tipos de conjuntos de datos puede trabajar el algoritmo.
- El método *buildClassifier*() construye el clasificador. *PMatrix* busca mediante computación evolutiva la matriz que mejor proyecte el conjunto de datos e indica al algoritmo base que construye el clasificador con el conjunto de datos proyectados. Además, se llevan a cabo las operaciones necesarias para generar los ficheros de salida, debiendo distinguir, entre otras cosas, si ha sido la primera vez que se ha invocado al método *buildClassifier*() en la actual ejecución del meta-algoritmo.
- El método *toString*() es invocado por Weka tras la primera invocación de *buildClassifier*() si y sólo si se ha indicado la opción *Output Model*[7], es decir, es invocado inmediatamente tras la construcción del modelo del clasificador base con el conjunto de datos completo. Desde este método, como es habitual, se imprime el modelo recién construido.
- Los métodos *setOptions*() y *getOptions*() sirven para establecer los valores que el usuario ha seleccionado a las diferentes variables y para mostrarlas en formato de línea de comando al principio de la información de la ejecución respectivamente.
- El método *listOptions*() es invocado cuando el meta-algoritmo es invocado erróneamente desde consola (*SimpleCLI*) y sirve para listar las opciones que pueden proporcionársele.
- El método *adaptar*() sirve para crear un conjunto de datos con un número de atributos (sin incluir la clase) igual al número de columnas seleccionado por el usuario en la ventana de diálogo del meta-algoritmo. Se le proporciona un conjunto de datos denominado *referencia*, ya que es la manera más sencilla y estándar de poder construir un conjunto de datos, puesto que *Weka* no proporciona funcionalidad (con visibilidad pública) para llevar a cabo esa tarea.

[7] Como ya se ha comentado, esta opción construye, independientemente del modo de test elegido, el modelo del algoritmo seleccionado con el conjunto de entrenamiento completo.

- El método *getInstanciaProy*() proyecta una instancia con la mejor matriz de pesos. Esta operación involucra, además, que se construya el conjunto de datos que la contendría, puesto que las instancias poseen un atributo con que referencia al conjunto de datos en que "están contenidas". Sin este atributo actualizado, el algoritmo base lanza una excepción al intentar clasificar dicha instancia.
- Los métodos *classifyInstance*() y *distributionForInstance*(), como ya se ha comentado, implementan la funcionalidad para clasificar una determinada instancia, sea nominal o no, respectivamente.
- El método *ejecutarGenético*() se encarga de inicializar todo lo necesario para llevar a cabo posteriormente el bucle de búsqueda (torneo, cruce, mutación y evaluación). El bucle acaba cuando no hay mejora durante el número de generaciones indicado.

Además, aunque no se ha indicado, para las variables cuyos valores son elegibles por el usuario desde la interfaz, aparte de la implementación obligada de sus accedentes y mutadores, se ha de implementar un método por cada variable denominado *nombreVariableTipText*() que devuelve una cadena de texto que será indicada como ayuda sobre el significado de esa variable en el algoritmo.

6.4 Ficheros de salida

El algoritmo *PMatrix* genera un conjunto de ficheros de salida en unos determinados directorios que pueden ser de gran utilidad en el proceso de aprendizaje, no sólo para entender mejor la búsqueda que se ha seguido, sino también para poder continuar el proceso de aprendizaje (se tienen disponibles los modelos generados, los espacios proyectados…).

Como se comentó en el apartado 6.3.4 para la clase *PMatrix*, la variable *numEjecuciones* lleva la cuenta del número de veces que el algoritmo se ha ejecutado (veces que se ha dado al botón *Start*). Para cada una de estas ejecuciones, se crea un directorio denominado *PMatrix Execution* <<*numEjecuciones*>>.

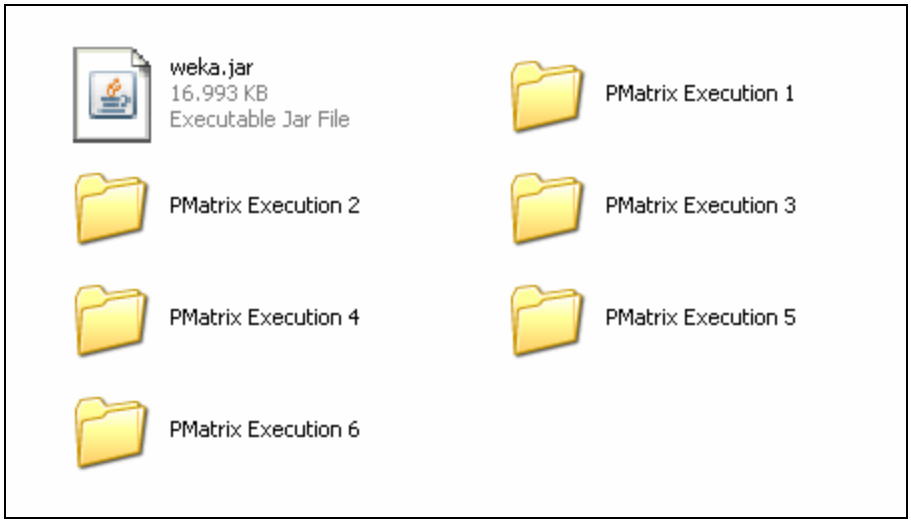

Ilustración 6.12 – Directorios generados por *PMatrix* en seis ejecuciones

En cada uno de estos directorios se almacenarán todos los ficheros relativos a dicha ejecución. Los ficheros serán los siguientes:

- El fichero *trace.txt* contiene, como su propio nombre indica, la traza de la búsqueda.

```
------ Original Projected (768 instances) - Sat May 09 01:29:05 CEST 2009 -----
Fitness: 75,78125       -0,0711 0,5751  ...  -0.6266    -0,1238
...
Fitness: 75,78125       -0,0711 0,5751  ...  -0,6266    -0,1238


------ Projected Training 1 (614 instances) - Sat May 09 01:29:26 CEST 2009 -----
Fitness: 74,42997       1,5513  -0,8212 ...  -0,3885    0,6970
...
Fitness: 76,54723       2,5568  -0,8212 ...  -0,2758    1,3412

...

------ Projected Training 5 (615 instances) - Sat May 09 01:31:00 CEST 2009 -----
Fitness: 75,44715       -0,0094 0,4308  ...  0,4204     0,7849
...
Fitness: 76,91057       -0,0125 0,4308  ...  0,4204     0,1986
```

Ilustración 6.13 – Contenido resumido del fichero *trace.txt*

Como puede observarse, el fichero corresponde a una ejecución del algoritmo con validación cruzada de cinco hojas como modo de test. El fichero ilustra como ha ido mejorando el fitness (y con qué matriz de pesos lo alcanzaba) en cada uno de los ficheros de entrenamiento: conjunto de datos original (puesto que la opción *Output Model* está activada) y otros cinco conjuntos de datos de entrenamiento que se corresponde cada uno con una iteración de la validación cruzada.

Para cada conjunto de entrenamiento, además, se antepone el número de instancias que lo componen y la fecha y hora en qué comenzó la búsqueda de su mejor matriz de proyección.

- El fichero *debug.txt* solamente se genera cuando la depuración está activada. Este fichero muestra la población para cada generación (al principio con todo detalle y posteriormente sólo el fitness de cada individuo), así como el principio y final de la ejecución de la búsqueda en cada conjunto de datos. Además, al inicio de cada búsqueda se muestran con qué valores se ejecutará para las diferentes variables.
- El fichero *debugP.txt* solamente se genera cuando la depuración está activada. Muestra todas las proyecciones realizadas en la ejecución, de conjuntos de datos y de instancias. Para las proyecciones de conjuntos de datos, se truncan las conjuntos de datos original y proyectado a las cinco primeras instancias para facilitar su seguimiento. Además, se indica, a modo de depuración, el número de atributos, el atributo clase y el número de instancias de todos los conjuntos de datos mostrados y el número de atributos y el atributo clase de todas las instancias mostradas.

```
------------------------------------------------------------------------
TRUNCATED DATASET TO PROJECT (768 inst, 9 attrib and class=8):
6,00000     148,00000       ...     50,00000        1,00000
...
0,00000     137,00000       ...     33,00000        1,00000

 MATRIX:
+0,57424        +0,00460        ...     +0,01158        -0,30309
...
+0,60400        +0,19665        ...     +0,83829        +0,77906

TRUNCATED PROJECTED DATASET (768 inst, 9 attrib and class=8):
231,23202       100,77331       ...     -64,93988       1,00000
...
220,19002       210,03068       ...     16,37097        1,00000
------------------------------------------------------------------------
------------------------------------------------------------------------
INSTANCE TO PROJECT (9 attrib and 8th is the class):
5,00000     168,00000       ...     41,00000        ?

 MATRIX:
-0,01246        +0,43077        ...     +0,26091        +0,72949
...
-1,86040        +1,09409        ...     +0,42040        +0,19855

PROJECTED INSTANCE (9 attrib and number 8 is the class):
-72,71770       562,61829       ...     -25,57127       ?
------------------------------------------------------------------------
```

Ilustración 6.14 – Contenido resumido del fichero *debugP.txt*

Además, se crearán dos directorios, denominados *arff* y *models*, que contendrán los conjuntos de datos proyectados y los modelos de cada conjunto de datos de entrenamiento y test con que se realiza el modo test, respectivamente.

Ilustración 6.15 – Ficheros de salida en una ejecución sin depuración

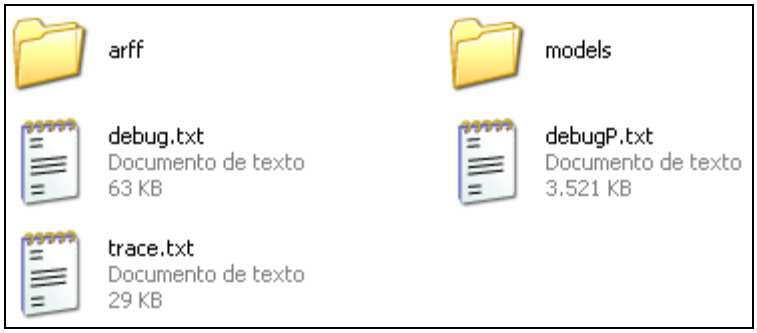

Ilustración 6.16 – Ficheros de salida en una ejecución con depuración

En el directorio *arff*, siguiendo el ejemplo, al haberse ejecutado seis conjuntos de datos de entrenamiento (el original y los cinco de la validación cruzada) y cinco conjuntos de datos de test (los cinco de la validación cruzada), habrá un total de 11 ficheros ARFF que contendrán la proyección de los mismos con sus mejores matrices de peso. Se ha supuesto que la mejor matriz de pesos para el conjunto de test es la misma que para su conjunto de entrenamiento correspondiente.

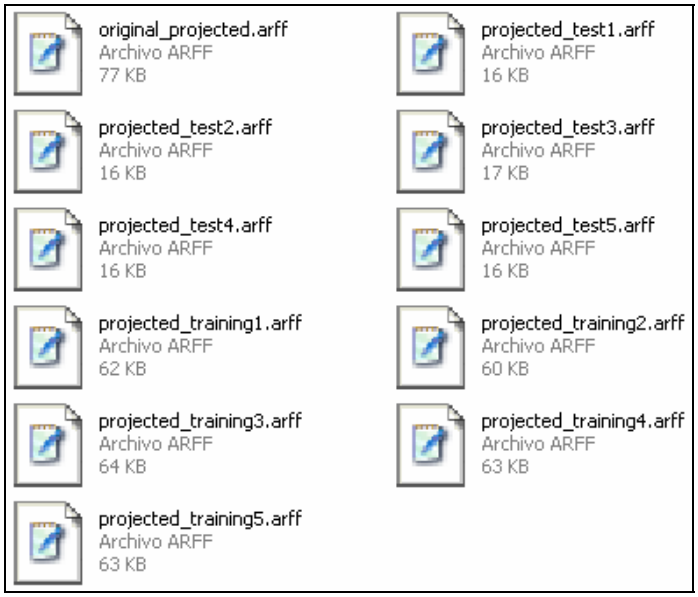

Ilustración 6.17 – Ficheros ARFF con los conjuntos de training y test proyectados

Los nombres de los atributos proyectados son *attrX*, siendo X un número natural iniciado en cero que se incrementa para cada atributo.

Al igual que con los ficheros ARFF, en el directorio *models*, siguiendo el ejemplo, al haberse ejecutado seis conjuntos de datos de entrenamiento (el original y los cinco de la validación cruzada) y cinco conjuntos de datos de test (los cinco de la validación cruzada), habrá un total de 11 ficheros de texto que contendrán los modelos construidos por el algoritmo base para dichos conjuntos de datos proyectados. Al igual que antes, se ha supuesto que la mejor matriz de pesos para el conjunto de test es la misma que para su conjunto de entrenamiento correspondiente.

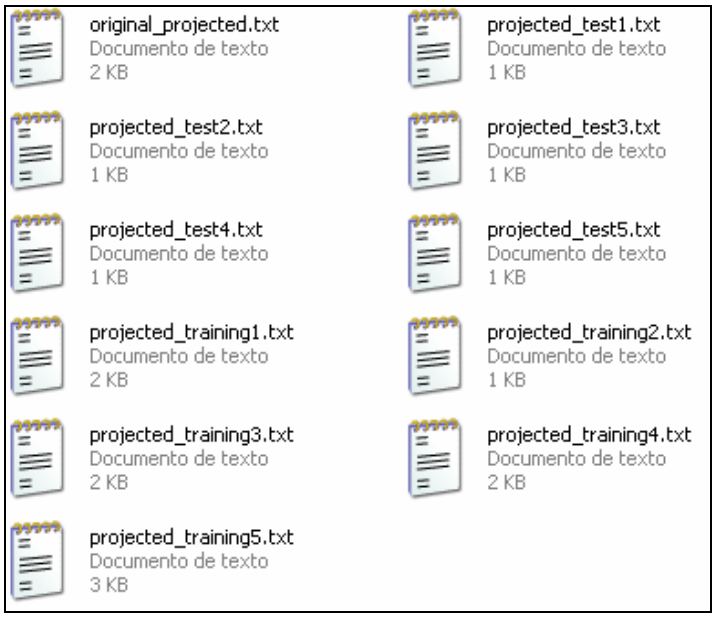

Ilustración 6.18 – Modelos de los conjuntos de training y test proyectados

Para cada conjunto de datos de training o test proyectado, se imprime su modelo y la matriz de pesos con que se ha proyectado. En este ejemplo, como se puede observar, el algoritmo base era el árbol J48 y se ilustra resumido el árbol que construye con los atributos proyectados:

```
J48 pruned tree
------------------

attr5 <= -93.860966: tested_positive (52.0/9.0)
attr5 > -93.860966
|   attr2 <= -305.021179
|   |   ...
|   attr2 > -305.021179
|   |   attr3 <= -114.078448
|   |   |   ...
|   |   attr3 > -114.078448: tested_negative (305.0/37.0)

Number of Leaves  :     12

Size of the tree :  23


Weights matrix:
-0,0711     0,5751   ...   -0,6266    -0,1238
```

Ilustración 6.19 – Contenido del fichero de modelo del conjunto original proyectado

7. Experimentación

En este capítulo se lleva a cabo la experimentación con el meta-algoritmo desarrollado. Se llevarán a cabo cuatro experimentos, tres de ellos con dominios sintéticos especialmente diseñados para poner a pruebas a *PMatrix*. En cada uno de ellos se describirá el conjunto de datos empleado, se experimentará sin y con *PMatrix*, para comparar la diferencia y se describirán las conclusiones de dicho experimento. Por último, se comentan las conclusiones globales de la experimentación.

7.1 Introducción

Una vez desarrollado el algoritmo *PMatrix*, la mejor forma de verificar su correcto funcionamiento es sometiéndole a un exhaustivo conjunto de experimentos.

En los siguientes apartados se describirán detalladamente cada uno de los experimentos realizados.

Salvo que se indique expresamente lo contrario, el modo de test empleado en los experimentos será validación cruzada con 10 hojas. Asimismo, el modo de depuración siempre estará activado.

Por último, el CD adjunto al proyecto contiene los ficheros de salida generados por el algoritmo desarrollado en cada una de las ejecuciones de cada experimento.

7.2 Experimento 1

En este apartado se explicará el primer experimento llevado a cabo donde se utiliza el dominio *rectas45*. En primer lugar, se describirá con detalle el dominio de entrada. En segundo lugar, se describirán los resultados que se obtienen sin el algoritmo desarrollado y, por último, cómo mejoran dichos resultados al aplicársele el meta-algoritmo desarrollado en este proyecto.

7.2.1 Dominio de entrada

Para este experimento se utilizará el domino denominado *rectas45*. Este dominio ha sido generado de manera artificial, tiene dos atributos (sin incluir el atributo clase) y 200 instancias. La clase es nominal y puede tomar dos valores (0 ó 1).

```
@relation rectas45
@attribute attr0 real
@attribute attr1 real
@attribute class {0,1}
@data
45.36107    46.484225   1
98.961915   98.599499   0
50.802933   50.983351   0
14.325776   15.473043   1
39.901731   39.848892   0
60.88634    60.553088   0
9.5112466   9.6310997   0
80.55274    80.570713   0
...
```

Ilustración 7.1 – Visualización resumida de fichero rectas45.arff

El fin principal que se ha perseguido al crear este dominio es imposibilitar a los algoritmos que trabajan con instancias vecinas (especialmente IB1) a llevar a cabo una buena clasificación. Esto se debe a que se ha pretendido que la instancia más cercana a cualquiera otra sea una instancia clasificada de manera diferente.

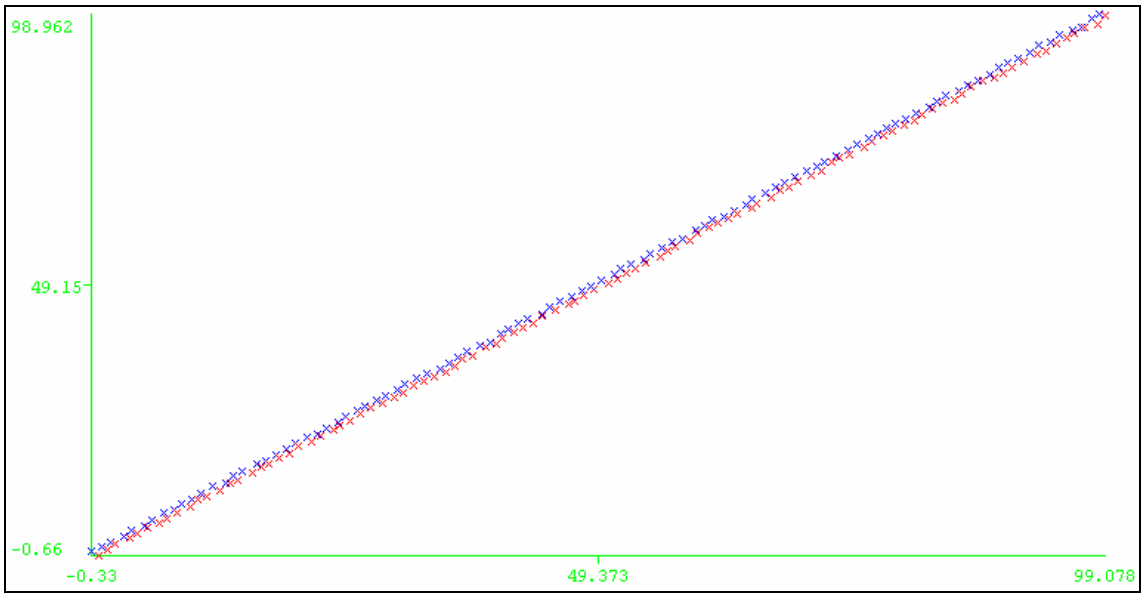

Ilustración 7.2 – Visualización bidimensional del conjunto de datos *rectas45*

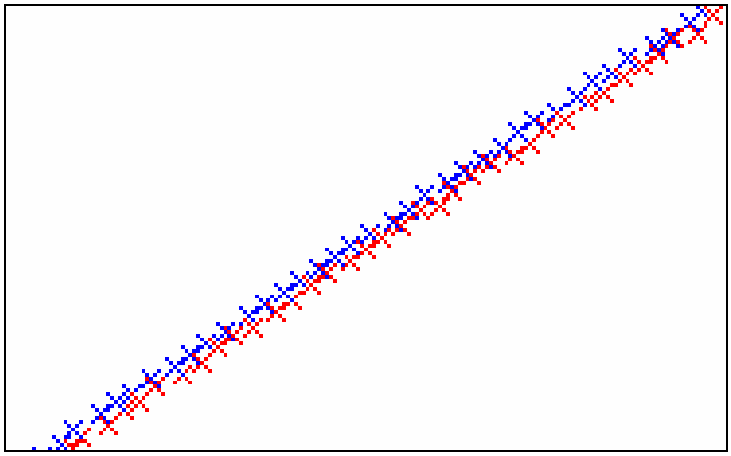

Ilustración 7.3 – Zoom de la visualización bidimensional de *rectas45*

Como puede observarse[1], los valores de los atributos van creciendo (aproximadamente entre cero y 100) siguiendo una pendiente de 45º (de ahí el nombre del dominio) y, para cada instancia, existe otra con unos valores para los atributos casi idénticos y, por el contrario, está clasificada de manera diferente. La clase es, pues, equilibrada, ya que 100 instancias están clasificadas con el valor cero y las 100 instancias restantes con el valor uno.

Cuando IB1 (que clasifica una instancia como perteneciente a la clase de la instancia más cercana) intenta clasificar este dominio los resultados son peores que el azar; no obstante, el algoritmo ZeroR, que trabaja con la premisa de clasificar a todas las instancias con la clase a la que más instancias están clasificadas, clasificaría correctamente al 50% de las instancias.

[1] Las cruces de color azul representan instancias etiquetados con la clase 0 y las cruces de color rojo representan instancias etiquetadas con la clase 1.

7.2.2 Experimentación sin PMatrix

En este apartado se intentará aprender de dicho conjunto de datos con el algoritmo IB1. En la tabla 7.1 se indica el porcentaje de instancias que ha conseguido clasificar correctamente:

Algoritmo utilizado	Aciertos
IB1	10.5%

Tabla 7.1 – Resultados del dominio rectas45 sin *PMatrix* (Exp. 1)

Como puede observarse en la tabla, el dominio sintético cumple su objetivo y la efectividad del algoritmo IB1 es realmente baja. En la ilustración 7.4 aparecen recuadradas las instancias mal clasificadas (prácticamente todas).

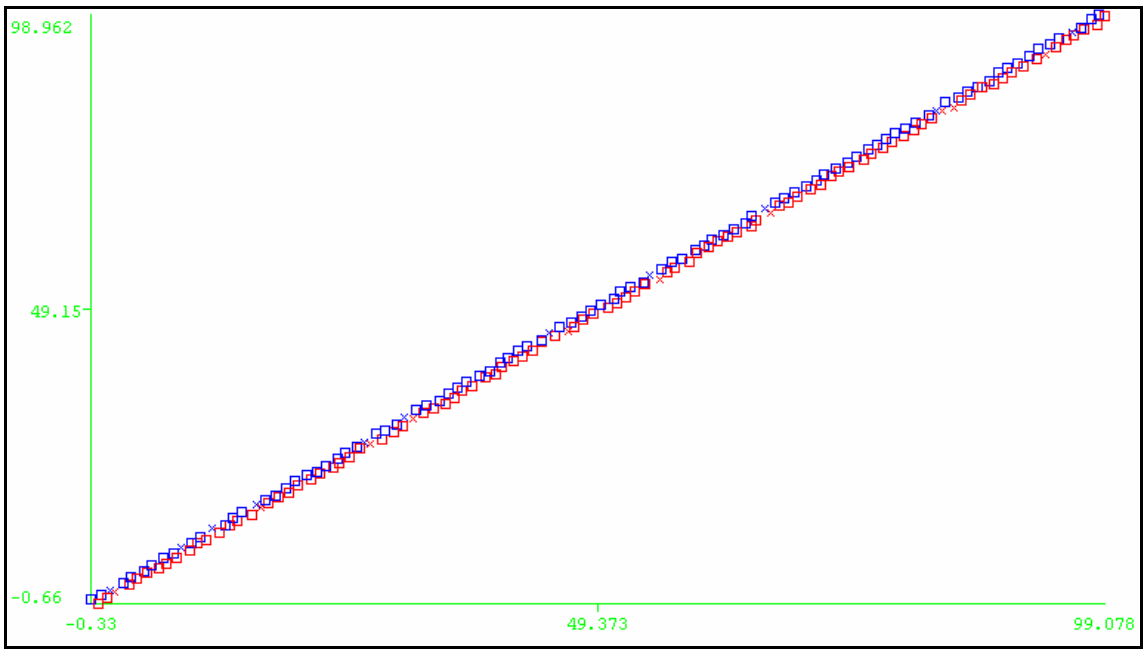

Ilustración 7.4 – Errores de clasificación con *IB1*

7.2.3 Experimentación con PMatrix

En este apartado se llevará a cabo la experimentación con el algoritmo desarrollado. En primer lugar, y puesto que dispone de numerosos parámetros configurables, se ejecutará con cada tipo de individuo (matiz completa, simétrica y diagonal) para, posteriormente, y con el tipo de individuo (matriz) más prometedor, llevar a cabo el ajuste de parámetros más cercano al óptimo para que se obtenga el mejor resultado posible.

Se denominará como 1.X a cada uno de los sucesivos experimentos, con distintos parámetros, que se llevarán a cabo; siendo X un número natural que se irá incrementando gradualmente. Además, para simplificar la legibilidad de los valores asignados a los 12 parámetros ajustables que ofrece *PMatrix*, se utilizará el siguiente formato tabular:

Parámetro	Valor
Clasificador base	
Cruce	
Constante de decremento	
Exponente	
Nº individuos en el cruce	
Número columnas (atributos)	
Nº generaciones sin mejora	
Tamaño de la población	
Prob. Mutación individuo	
Prob. Mutación entrada (matriz)	
Tamaño del torneo	
Tipo Individuo	

Tabla 7.2 – Formato de visualización de valores de parámetros de *PMatrix*

Por último, y cuando resulte beneficioso, se visualizarán los datos proyectados para observar cómo han sido proyectados con la mejor matriz encontrada.

Una vez explicado como será el proceso de experimentación con el algoritmo desarrollado se ejecutará el algoritmo con el primer tipo de individuo (matriz completa) y el resto de parámetros tendrán valores que garanticen una búsqueda lo más completa posible. En resumen, para el experimento 1.1 se utilizarán los siguientes valores:

Parámetro	Valor
Clasificador base	IB1
Cruce	Activado
Constante de decremento	0.93
Exponente	1.0
Nº individuos en el cruce	6
Número columnas (atributos)	-1
Nº generaciones sin mejora	4
Tamaño de la población	8
Prob. Mutación individuo	65
Prob. Mutación entrada (matriz)	15
Tamaño del torneo	3
Tipo Individuo	1

Tabla 7.3 – Valores de los parámetros de *PMatrix* en el Exp. 1.1

El porcentaje de aciertos obtenido con la matriz completa es del 99%. En la ilustración 7.5 se muestra el conjunto de datos completo proyectado.

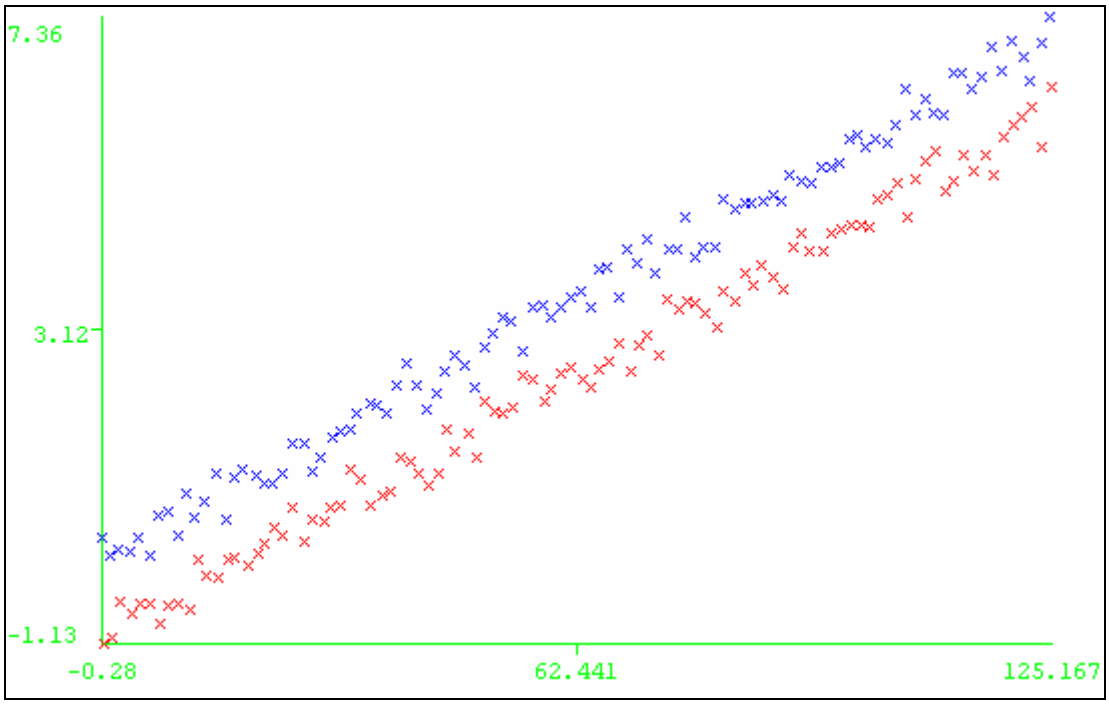

Ilustración 7.5 – Instancias proyectadas con matriz completa (Exp. 1.1)

En el experimento 1.2 (con matrices simétricas, tipo de individuo 2) se obtiene un porcentaje de aciertos del 99.5%. En la ilustración 7.6 se muestra el conjunto de datos completo proyectado, se puede observar, a pesar de que los dominios de las gráficas son diferentes, que la matriz simétrica ha separado las instancias de cada clase más que la matriz completa:

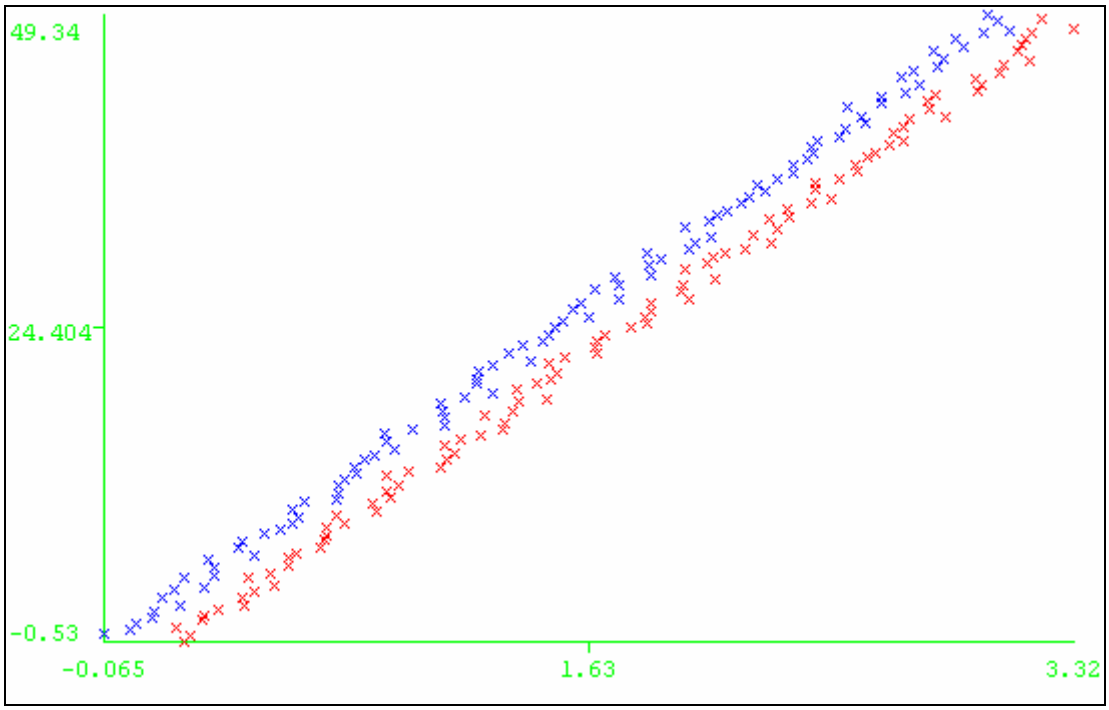

Ilustración 7.6 – Instancias proyectadas con matriz simétrica (Exp. 1.2)

En el experimento 1.3 se experimenta con matrices diagonales (tipo de individuo 3), con las que se obtiene un porcentaje de aciertos del 10.5%. La matriz diagonal falla estrepitosamente debido a que sus proyecciones son equivalentes a multiplicar cada atributo por un coeficiente, de tal modo que se separan los datos de distintas clases en la misma medida que los de la misma clase, por tanto, no puede conseguirse que la instancia más cercana a otra sea una de su misma clase; de hecho, el porcentaje de aciertos es idéntico al que se obtiene con el clasificador base (*IB1*). Para conseguir mejorar, se necesitan rotar y, posteriormente, separar los datos, y la rotación es algo que no permiten las matrices diagonales, pero sí las simétricas y completas.

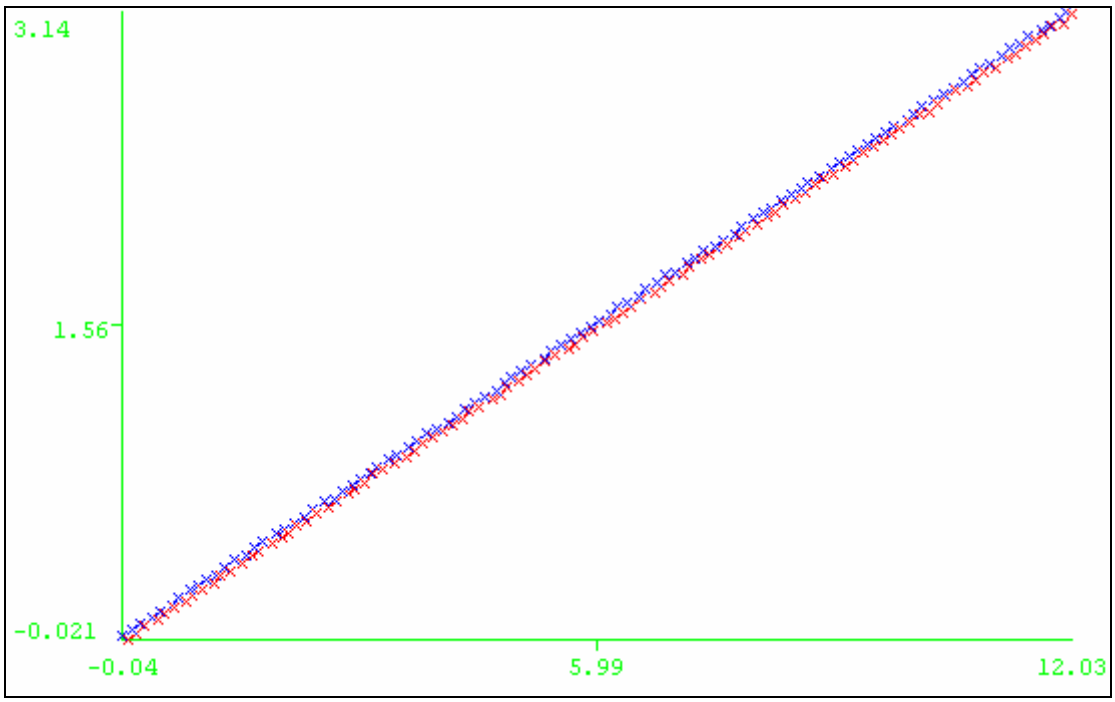

Ilustración 7.7 – Instancias proyectadas con matriz diagonal (Exp. 1.3)

En la tabla 7.4 se muestran, a modo de resumen, los resultados obtenidos con los distintos tipos de matrices/individuos:

Tipo de Matriz	% Aciertos
Identidad[2]	10.5%
Completa	99%
Simétrica	99.5%
Diagonal	10.5%

Tabla 7.4 – Comparativa de los resultados de los Exp. 1.1 a 1.3

En este dominio, la matriz simétrica ha resultado ser la más efectiva, por tanto, los sucesivos experimentos irán encaminados a llevar a cabo una búsqueda más potente (aumentando los valores de atributos clave) con el tipo de matriz indicado para conseguir una tasa de aciertos del 100%. Así, el experimento 1.4 se llevará a cabo con

[2] Corresponde al primer experimento, realizado con IB1 sin PMatrix.

los siguientes valores (donde aparecen en negrita aquellos parámetros modificados con respecto al último experimento):

Parámetro	Valor
Clasificador base	IB1
Cruce	Activado
Constante de decremento	**0.95**
Exponente	1.0
Nº individuos en el cruce	**30**
Número columnas (atributos)	-1
Nº generaciones sin mejora	**8**
Tamaño de la población	**75**
Prob. mutación individuo	**70**
Prob. mutación entrada (matriz)	**25**
Tamaño del torneo	**15**
Tipo Individuo	**2**

Tabla 7.5 – Valores de los parámetros de *PMatrix* en el Exp. 1.4

La ejecución arroja un resultado del 99% de instancias clasificadas correctamente, idéntico al resultado del experimento 1.1. Experimentos similares no mostraron mejoras respecto al resultado del experimento 1.2. Por tanto, puesto que el conjunto de datos está compuesto por 200 instancias, es una sola instancia la que no puede ser clasificada correctamente y la que provoca que el meta-algoritmo desarrollado no arroje un resultado perfecto.

7.2.4 Conclusiones

Concluido este primer experimento, se muestra que el algoritmo *PMatrix* funciona correctamente.

El dominio *rectas45* es un dominio sintético que ha sido generado para verificar su correcto funcionamiento. *IB1* logra clasificar solamente un 10.5% de las instancias correctamente, pero cuando *PMatrix* lo utiliza como clasificador base, el resultado asciende al 99.5%.

Resulta también sorprendente que con una población tan pequeña (en los experimentos 1.1 a 1.3 de ocho individuos) se consigan unos resultados tan positivos. Además, la ejecución del algoritmo es muy rápida, a modo de ejemplo, el experimento 1.4 (con una población de 75 individuos) tardó 37 segundos en un Pentium IV a 3.2 GHz con Windows XP.

7.3 Experimento 2

En este apartado se explicará el primer experimento llevado a cabo donde se utiliza el dominio *rectas0*. En primer lugar, se describirá con detalle el dominio de entrada. En segundo lugar, se describirán los resultados que se obtienen sin el algoritmo desarrollado y, por último, cómo mejoran dichos resultados al aplicársele el meta-algoritmo desarrollado en este proyecto.

7.3.1 Dominio de entrada

Para este experimento se utilizará el domino denominado *rectas0*. Este dominio ha sido, al igual que el anterior, generado sintéticamente, tiene dos atributos numéricos (sin incluir el atributo clase) y 200 instancias. La clase es nominal y puede tomar dos valores (0 ó 1).

```
@relation rectas0
@attribute attr0 real
@attribute attr1 real
@attribute class {0,1}
@data
93.031527   0.14009185  0
111.35621   0.84555198  1
115.8431    0.66143446  1
37.891979   0.79095397  1
17.942178   0.59727287  1
64.713301   0.37464951  1
76.230393   0.83701949  1
57.431451   0.10886285  0
...
```

Ilustración 7.8 – Visualización resumida de fichero rectas0.arff

El fin principal que se ha perseguido al crear este dominio es mostrar que con una matriz diagonal es suficiente para clasificar (óptimamente) este conjunto de datos, ya que con un simple cambio de escala puede conseguirse.

En la ilustración 7.9 se observa cómo teniendo en cuenta la distancia real entre las instancias, el aprendizaje no puede ser óptimo con un algoritmo basado en el vecino más cercano, puesto que al igual que en el experimento 1, los datos más cercanos pertenecen a clases distintas.

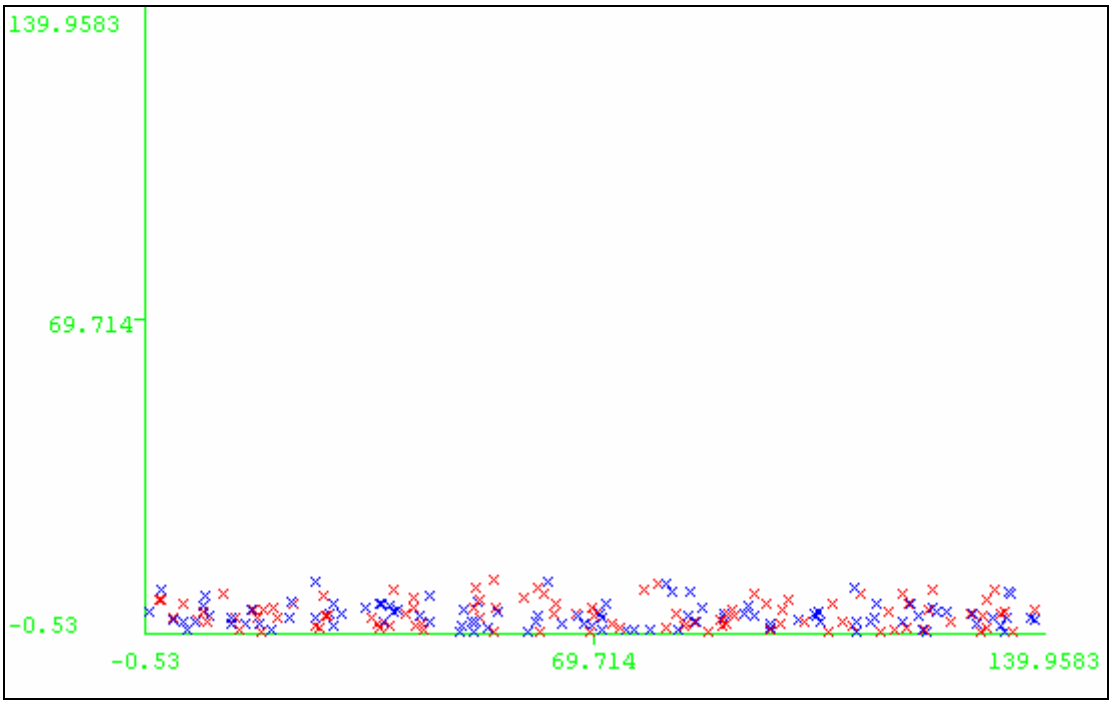

Ilustración 7.9 – Visualización bidimensional del conjunto de datos *rectas0*

Con un cambio de escala, o lo que es lo mismo, multiplicando la coordenada "Y" para separar los datos en la dirección vertical sería suficiente para que el aprendizaje por parte de un algoritmo basado en el vecino más cercano pudiera ser mejorado. Esto se puede hacer con matrices diagonales, que equivalen a un cambio de escala.

La ilustración 7.10 muestra el conjunto de datos en distinta escala y puede observarse cómo utilizando la distancia euclídea clásica, *IB1* podría aprender óptimamente el conjunto de datos.

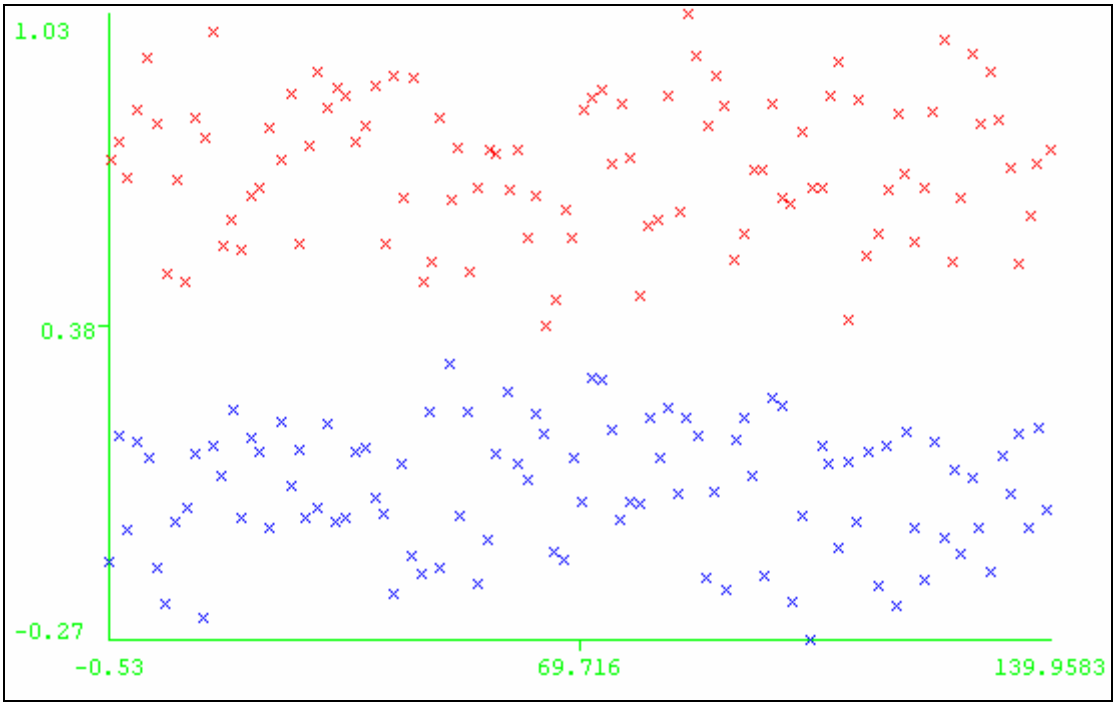

Ilustración 7.10 – Visualización del conjunto de datos *rectas0* escalado

7.3.2 Experimentación sin PMatrix

En este apartado se intentará aprender de dicho conjunto de datos con un algoritmo basado en el vecino más cercano. Sin embargo, el algoritmo *IB1* implementado en *Weka* logra clasificar correctamente el 100% de las instancias debido a que utiliza la distancia euclídea normalizada (la cual produce el cambio de escala requerido en este dominio). Con el objetivo de utilizar dicho algoritmo con la distancia euclídea no normalizada y puesto que *IB1* no permite dicha configuración, se utilizará como algoritmo base *IBk* con un vecino, siendo *LinearNNSearch* el algoritmo de búsqueda del vecino más cercano y configurando su función de distancia como la distancia euclídea no normalizada.

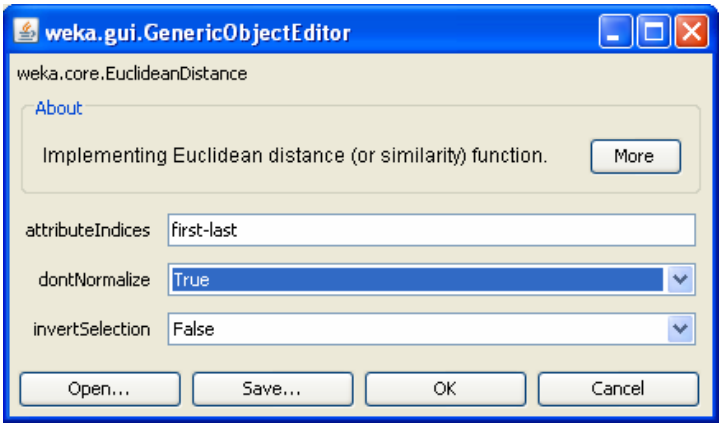

Ilustración 7.11 – Configuración de la distancia euclídea no normalizada

En la tabla 7.6 se indica el porcentaje de instancias que ha conseguido clasificar correctamente:

Algoritmo utilizado	Aciertos
IBk	9.5%

Tabla 7.6 – Resultados del dominio rectas0 sin *PMatrix* (Exp. 2)

Como puede observarse en la tabla 7.6, el dominio sintético cumple su objetivo y la efectividad del algoritmo *IBk* es realmente baja.

7.3.3 Experimentación con PMatrix

En este apartado se llevará a cabo la experimentación con el algoritmo desarrollado. Al contrario que en el experimento 1, sólo se ejecutará para la matriz diagonal y, posteriormente, llevar a cabo el ajuste de parámetros más cercano al óptimo para que se obtenga el mejor resultado posible.

De nuevo, se denominará como 2.X a cada uno de los sucesivos experimentos, con distintos parámetros, que se llevarán a cabo; siendo X un número natural que se irá incrementando gradualmente. Los valores de los parámetros se mostrarán en el formato tabular utilizado en el experimento 1.

Por último, y cuando resulte beneficioso, se visualizarán los datos proyectados para observar cómo han sido proyectados con la mejor matriz encontrada.

Una vez explicado como será el proceso de experimentación con el algoritmo desarrollado se ejecutará el algoritmo con el primer tipo de individuo (matriz completa) y el resto de parámetros tendrán valores que garanticen una búsqueda lo más completa posible. En resumen, para el experimento 2.1 se utilizarán los siguientes valores:

Parámetro	Valor
Clasificador base	IBk
Cruce	Activado
Constante de decremento	0.95
Exponente	1.0
Nº individuos en el cruce	8
Número columnas (atributos)	-1
Nº generaciones sin mejora	8
Tamaño de la población	25
Prob. mutación individuo	65
Prob. mutación entrada (matriz)	20
Tamaño del torneo	9
Tipo Individuo	3

Tabla 7.7 – Valores de los parámetros de *PMatrix* en el Exp. 2.1

La ejecución arroja un resultado del 89% de instancias clasificadas correctamente. A continuación, se realizará un nuevo experimento en el que se aumentarán ciertos parámetros para potenciar la búsqueda y encontrar un resultado mejor. Los valores que se utilizarán para el experimento 2.2 serán:

Parámetro	Valor
Clasificador base	IBk
Cruce	Activado
Constante de decremento	0.95
Exponente	1.0
Nº individuos en el cruce	35
Número columnas (atributos)	-1
Nº generaciones sin mejora	12
Tamaño de la población	100
Prob. mutación individuo	65
Prob. mutación entrada (matriz)	20
Tamaño del torneo	18
Tipo Individuo	3

Tabla 7.8 – Valores de los parámetros de *PMatrix* en el Exp. 2.2

Con esta configuración, la ejecución arroja un resultado perfecto; se han clasificado correctamente el 100% de las instancias.

En la ilustración 7.12 puede observarse cómo quedan los datos del conjunto original proyectados con la matriz diagonal.

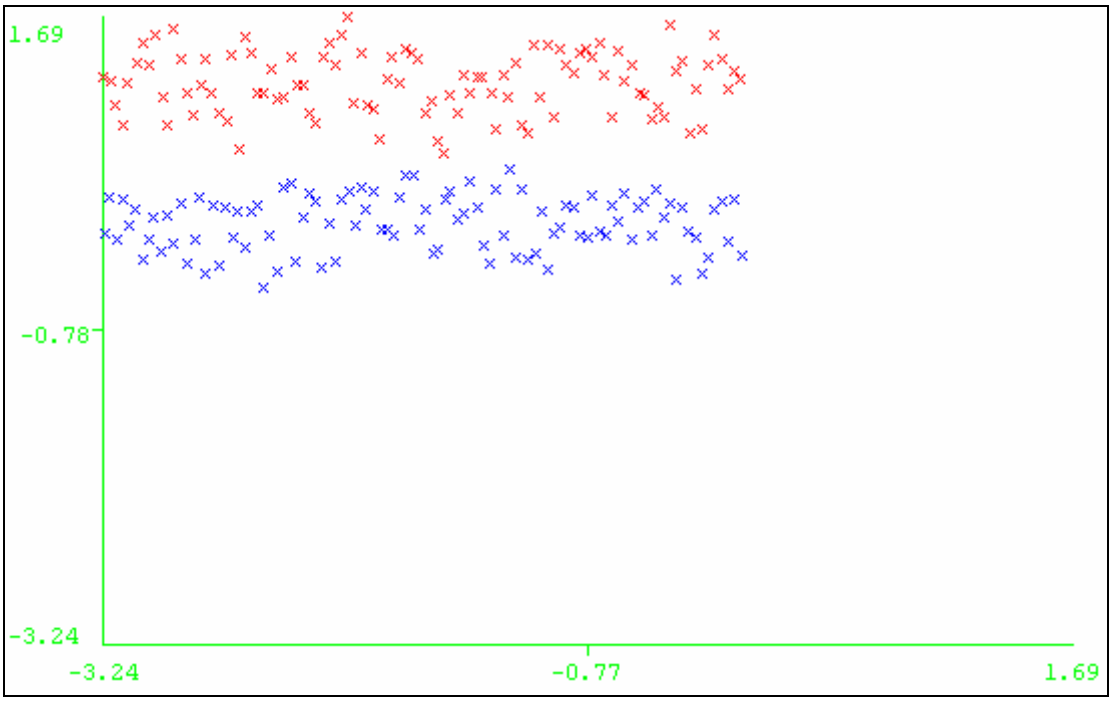

Ilustración 7.12 – Visualización escalada del conjunto de datos proyectado

7.3.4 Conclusiones

Este experimento ha proporcionado una nueva y fuerte evidencia de la utilidad de *PMatrix* sobre ciertos conjuntos de datos, en este caso, aquellos en los que un simple cambio de escala es necesario para conseguir un mejor aprendizaje (en este caso dicho aprendizaje es óptimo).

De nuevo, el algoritmo ha realizado el aprendizaje en un tiempo muy reducido, siendo inferior a un minuto en un Pentium IV a 3.2 GHz con Windows XP, a pesar del considerable tamaño de la población que maneja el algoritmo evolutivo.

7.4 Experimento 3

En este apartado se explicará el primer experimento llevado a cabo donde se utiliza el dominio *diabetes*. En primer lugar, se describirá con detalle el dominio de entrada. En segundo lugar, se describirán los resultados que se obtienen sin el algoritmo desarrollado y, por último, cómo mejoran dichos resultados al aplicársele el meta-algoritmo desarrollado en este proyecto.

7.4.1 Dominio de entrada

Para este experimento se utilizará el domino denominado *diabetes*. Se trata de un dominio real elaborado en 1990 por el National Institute of Diabetes and Digestive and Kidney Diseases (Instituto Nacional de Diabetes y Enfermedades Digestivas y Renales) con sede en Maryland, USA. Este será, por tanto, el primer experimento en el que se aplicará el meta-algoritmo desarrollado a un conjunto de datos real, en contraposición a los dos experimentos anteriores en que se utilizaron conjuntos de datos sintéticos para demostrar el buen funcionamiento de *PMatrix*.

El conjunto de datos contiene 768 instancias[3] con ocho atributos cada una (sin incluir la clase). La clase es nominal y representa el resultado del test de diabetes, positivo o negativo en dicha persona (instancia). Los nueve atributos son los siguientes:

- **preg**: Número de veces que la paciente ha estado embarazada
- **plas**: Concentración de glucosa en plasma tras 2 horas del test de tolerancia a la glucosa.
- **pres**: Presión de la sangre en diástole.
- **skin**: Grosura de la piel en el tríceps (en milímetros).
- **insu**: Nivel de insulina en sangre a 2 horas del test.
- **mass**: Índice de masa corporal (IMC)[4].
- **pedi**: Función genealógica de la diabetes.
- **age**: Edad (en años).

[3] Corresponden a 768 mujeres de, al menos, 21 años.

[4] IMC = $\frac{Peso}{Altura^2}$

```
@relation pima_diabetes
@attribute 'preg' real
@attribute 'plas' real
@attribute 'pres' real
@attribute 'skin' real
@attribute 'insu' real
@attribute 'mass' real
@attribute 'pedi' real
@attribute 'age' real
@attribute 'class' { tested_negative, tested_positive}
@data
6,148,72,35,0,33.6,0.627,50,tested_positive
1,85,66,29,0,26.6,0.351,31,tested_negative
8,183,64,0,0,23.3,0.672,32,tested_positive
1,89,66,23,94,28.1,0.167,21,tested_negative
0,137,40,35,168,43.1,2.288,33,tested_positive
5,116,74,0,0,25.6,0.201,30,tested_negative
...
```

Ilustración 7.13 – Visualización resumida de fichero diabetes.arff

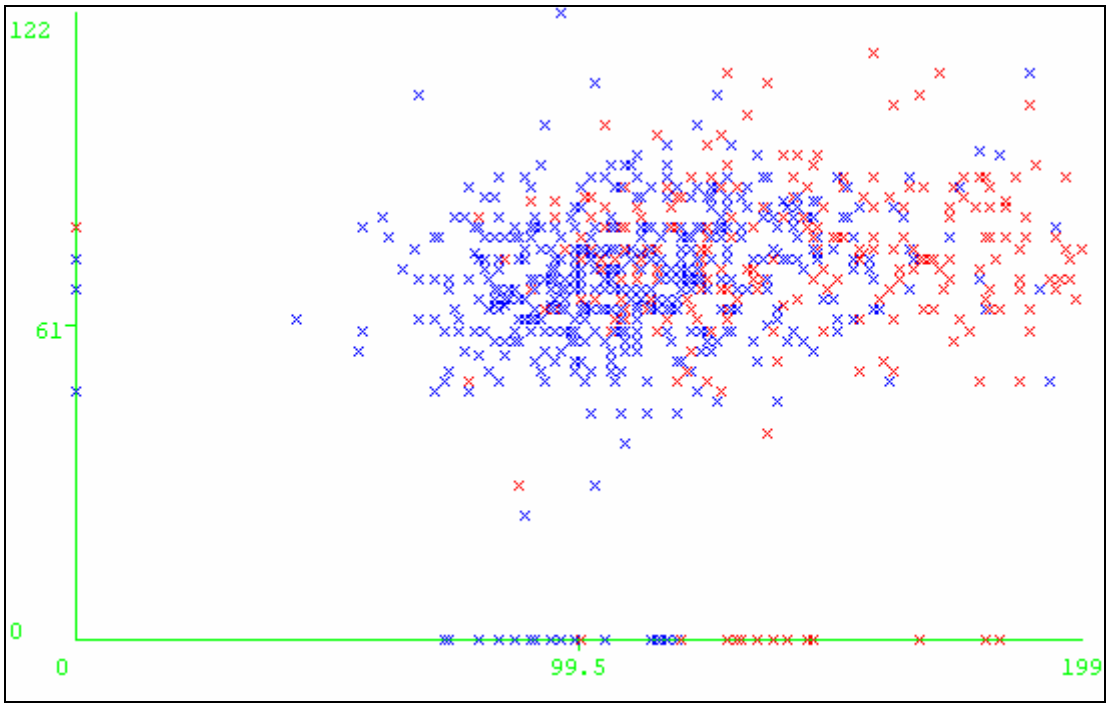

Ilustración 7.14 – Visualización de los atributos plas y preg

Como puede observarse en la ilustración 7.14, los valores de los atributos para las distintas clases están muy solapados, lo que hace especialmente complicado su aprendizaje (con un clasificador lineal).

7.4.2 Experimentación sin PMatrix

En este apartado se intentará aprender de dicho conjunto de datos con el algoritmo J48. En la tabla 7.9 se indica el porcentaje de instancias que ha conseguido clasificar correctamente:

Algoritmo utilizado	Aciertos
J48	73.8281%

Tabla 7.9 – Resultados del dominio rectas45 sin *PMatrix* (Exp. 3)

Como puede observarse en la tabla 7.9, el conjunto de datos es aprendido razonablemente bien por J48.

7.4.3 Experimentación con PMatrix

En este apartado se llevará a cabo la experimentación con el algoritmo desarrollado. Al igual que en el experimento 1, se ejecutará para cada tipo de matriz y, posteriormente, con el tipo de matriz más prometedora, llevar a cabo un ajuste de parámetros más potente para que se obtenga el mejor resultado posible.

De nuevo, se denominará como 3.X a cada uno de los sucesivos experimentos, con distintos parámetros, que se llevarán a cabo; siendo X un número natural que se irá incrementando gradualmente. Los valores de los parámetros se mostrarán en el formato tabular utilizado en los experimentos anteriores.

Por último, y cuando resulte beneficioso, se visualizarán los datos proyectados para observar cómo han sido proyectados con la mejor matriz encontrada.

Una vez explicado como será el proceso de experimentación con el algoritmo desarrollado se ejecutará el algoritmo con el primer tipo de individuo (matriz completa) y el resto de parámetros tendrán valores que garanticen una búsqueda lo más completa posible. En resumen, para el experimento 3.1 se utilizarán los siguientes valores:

Parámetro	Valor
Clasificador base	J48
Cruce	Activado
Constante de decremento	0.93
Exponente	1.0
Nº individuos en el cruce	8
Número columnas (atributos)	-1
Nº generaciones sin mejora	7
Tamaño de la población	15
Prob. mutación individuo	65
Prob. mutación entrada (matriz)	20
Tamaño del torneo	6

Tipo Individuo	**1**

Tabla 7.10 – Valores de los parámetros de *PMatrix* en el Exp. 3.1

PMatrix clasifica correctamente al 72.1354% de las instancias con la configuración de la tabla 7.10. Para el experimento 3.2, con matrices simétricas, se utilizarán los siguientes valores:

Parámetro	Valor
Clasificador base	J48
Cruce	Activado
Constante de decremento	0.93
Exponente	1.0
Nº individuos en el cruce	8
Número columnas (atributos)	-1
Nº generaciones sin mejora	7
Tamaño de la población	15
Prob. mutación individuo	65
Prob. mutación entrada (matriz)	20
Tamaño del torneo	6
Tipo Individuo	**2**

Tabla 7.11 – Valores de los parámetros de *PMatrix* en el Exp. 3.2

Con la configuración de la tabla 7.11, *PMatrix* clasifica correctamente al 73.0469% de las instancias. Para el experimento 3.3, con matrices diagonales, se utilizarán los siguientes valores:

Parámetro	Valor
Clasificador base	J48
Cruce	Activado
Constante de decremento	0.93
Exponente	1.0
Nº individuos en el cruce	8
Número columnas (atributos)	-1
Nº generaciones sin mejora	7
Tamaño de la población	15
Prob. mutación individuo	65
Prob. mutación entrada (matriz)	20
Tamaño del torneo	6
Tipo Individuo	**3**

Tabla 7.12 – Valores de los parámetros de *PMatrix* en el Exp. 3.3

PMatrix clasifica correctamente al 74.2188% de las instancias con la configuración de la tabla 7.12. Por tanto, la mejor matriz para este dominio con los experimentos 3.1 a 3.3 resulta ser la matriz diagonal.

Tipo de Matriz	% Aciertos
Identidad[5]	73.8281%
Completa	72.1354%
Simétrica	73.0469%
Diagonal	74.2188%

Tabla 7.13 – Comparativa de los resultados de los Exp. 3.1 a 3.3

A continuación, se realizará un nuevo experimento en el que se modificarán, respecto al experimento anterior, ciertos parámetros (mostrados en negrita) para potenciar la búsqueda y encontrar un resultado mejor. Los valores que se utilizarán para el experimento 3.4 serán los siguientes:

Parámetro	Valor
Clasificador base	J48
Cruce	Activado
Constante de decremento	**0.96**
Exponente	1.0
Nº individuos en el cruce	**35**
Número columnas (atributos)	-1
Nº generaciones sin mejora	**12**
Tamaño de la población	**50**
Prob. mutación individuo	65
Prob. mutación entrada (matriz)	20
Tamaño del torneo	**18**
Tipo Individuo	3

Tabla 7.14 – Valores de los parámetros de *PMatrix* en el Exp. 3.4

Con esta configuración, la ejecución arroja un resultado del 74.0885%, inferior a la conseguida en el experimento 3.3. La configuración de parámetros especificada en la tabla 7.15 arrojó un resultado de 76.6927% de instancias clasificadas correctamente.

Parámetro	Valor
Clasificador base	J48
Cruce	Activado
Constante de decremento	0.96
Exponente	**2.0**
Nº individuos en el cruce	**25**
Número columnas (atributos)	**4**
Nº generaciones sin mejora	12
Tamaño de la población	**40**
Prob. mutación individuo	**50**
Prob. mutación entrada (matriz)	**12**
Tamaño del torneo	**10**
Tipo Individuo	**1**

Tabla 7.15 – Valores de los parámetros de *PMatrix* en el Exp. 3.5

[5] Corresponde al primer experimento, realizado con J48 sin *PMatrix*.

En la ilustración 7.15 puede observarse un gráfico con la evolución del fitness para las proyecciones del conjunto de datos original.

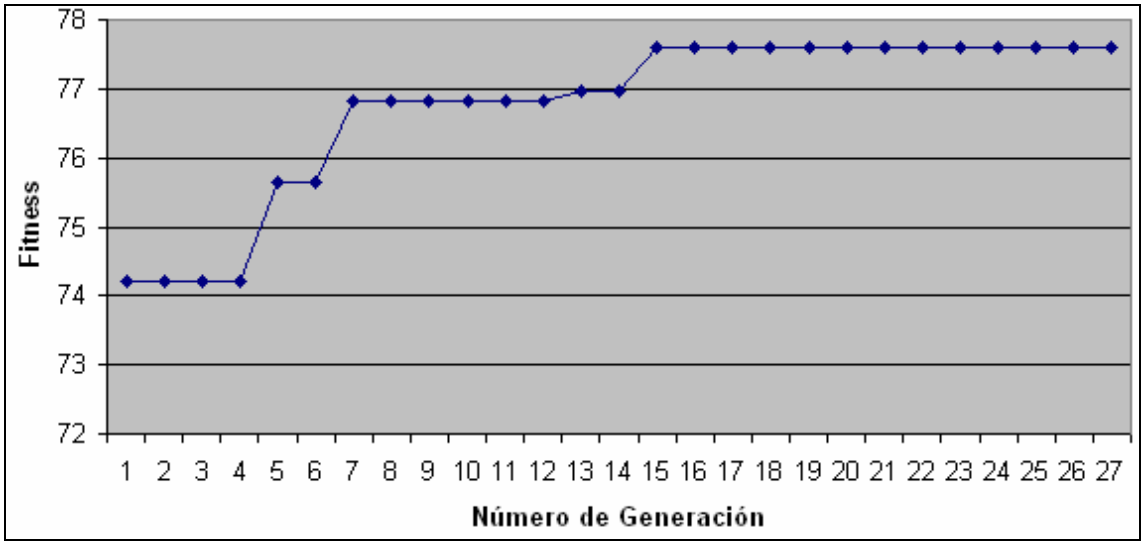

Ilustración 7.15 – Evolución del fitness del conjunto de datos original proyectado

En la tabla 7.16 se muestra una comparativa con los resultados de los algoritmos con configuraciones avanzadas:

Tipo de Matriz	% Aciertos
Identidad[6]	73.8281%
Diagonal (Exp. 3.4)	74.0885%
Completa (Exp. 3.5)	76.6927%

Tabla 7.16 – Comparativa de los resultados de los Exp. 3.4 y 3.5

7.4.4 Conclusiones

En este dominio la mejora que ha supuesto *PMatrix* es del 3.88%, considerablemente inferior si es comparada con la mejora de los experimentos anteriores. Sin embargo, puede considerarse un buen resultado puesto que es un dominio real cuyos resultados son difíciles de mejorar.

En este experimento el meta-algoritmo ha necesitado un mayor tiempo; habiendo tardado 22 minutos (en un Pentium IV a 3.2 GHz con Windows XP) en llevar a cabo el experimento 3.4 y 17 minutos para el experimento 3.5.

[6] Corresponde al primer experimento, realizado con J48 sin *PMatrix*.

7.5 Experimento 4

En este apartado se explicará el primer experimento llevado a cabo donde se utiliza el dominio *aleatorios100*. En primer lugar, se describirá con detalle el dominio de entrada. En segundo lugar, se describirán los resultados que se obtienen sin el algoritmo desarrollado y, por último, cómo mejoran dichos resultados al aplicársele el meta-algoritmo desarrollado en este proyecto.

7.5.1 Dominio de entrada

Para este experimento se utilizará el domino denominado *aleatorios100*. Este dominio ha sido, al igual que los dos primeros, generado sintéticamente, tiene cuatro atributos numéricos (sin incluir el atributo clase) y 200 instancias. La clase es nominal y puede tomar dos valores (0 ó 1).

```
@relation aleatorios
@attribute attr0 real
@attribute attr1 real
@attribute attr2 real
@attribute attr3 real
@attribute class {0,1}
@data
0.302280287 0.287512198 29.9934026  96.2496596  1
0.485135682 0.456737338 31.2168798  27.6336284  1
0.014512753 0.869219617 39.0604951  9.180895    0
0.168479671 0.440624854 52.9817369  48.3006925  0
0.005177929 0.589641721 55.660838   52.6266035  0
0.830652942 0.010290971 19.4512272  49.5153647  1
0.714778326 0.077720084 34.6127464  78.2829677  1
...
```

Ilustración 7.16 – Visualización resumida de fichero *aleatorio100.arff*

El tercer y cuarto atributos son irrelevantes y están multiplicados por 100. En este caso trabajaremos con matrices de cuatro filas y dos columnas, que son capaces de eliminar (en principio) los dos atributos irrelevantes.

En la ilustración 7.17 aparecen confrontados los valores de los atributos irrelevantes, el atributo dos y el atributo tres. Como puede observarse, estos atributos sin imposibles de aprender.

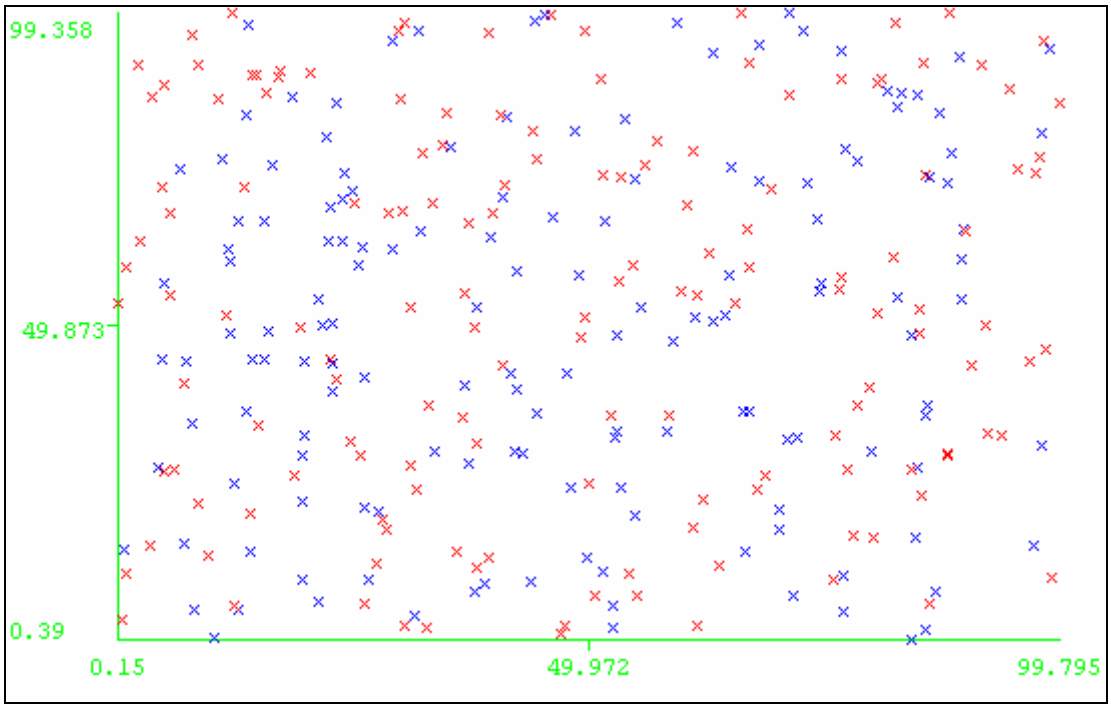

Ilustración 7.17 – Visualización de los atributos 2 y 3 de aleatorio100.arff

Por el contrario, en la ilustración 7.18 aparecen confrontados los valores de los atributos cero y uno, que deberían ser rotados 45 grados para ser aún mejor aprendidos por una clasificador como *J48*. La razón es que *J48* es un algoritmo que divide el espacio mediante rectángulos paralelos a los ejes. Por ejemplo, para aproximar la zona triangular inferior que representa la clase roja (ver ilustración 7.18), *J48* necesita construir muchos rectángulos estrechos, paralelos al eje "Y", que van siguiendo la bisectriz de 45 grados. Si los datos se rotan 45 grados hacia la izquierda, la clasificación se convierte en trivial, porque todos los datos a la izquierda del eje "Y" pertenecerán a la clase azul y los de la derecha, a la clase roja.

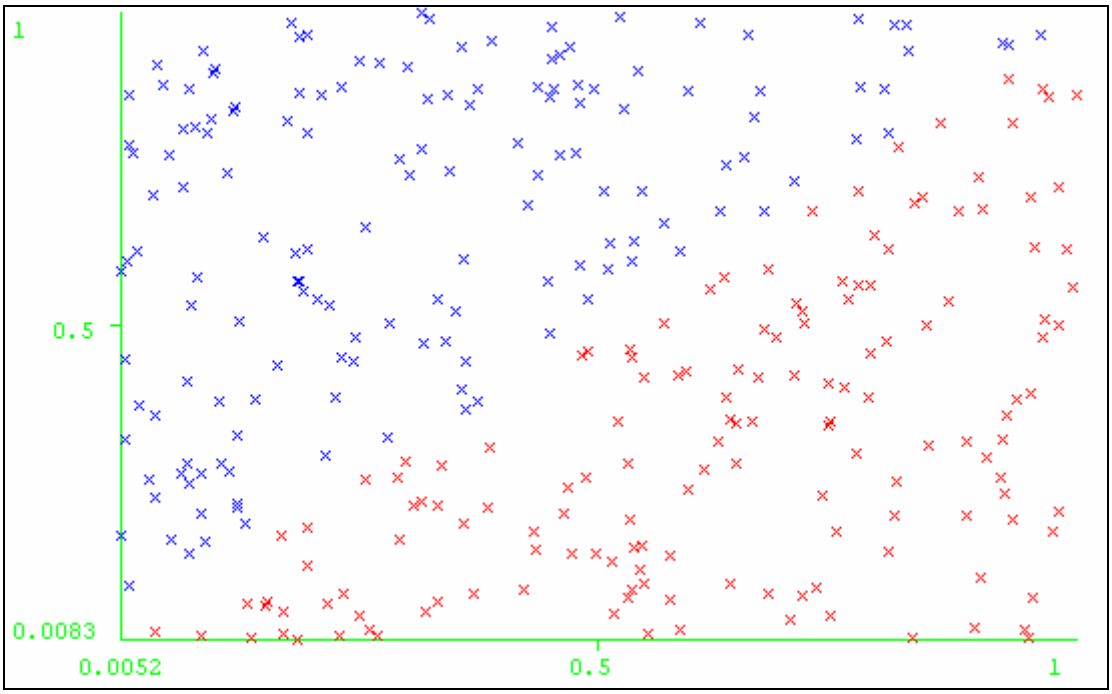

Ilustración 7.18 – Visualización de los atributos 0 y 1 de aleatorio100.arff

7.5.2 Experimentación sin PMatrix

En este apartado se intentará aprender de dicho conjunto de datos con el algoritmo J48. En la tabla 7.17 se indica el porcentaje de instancias que ha conseguido clasificar correctamente:

Algoritmo utilizado	Aciertos
J48	93.3333%

Tabla 7.17 – Resultados del dominio aleatorios100 sin *PMatrix* (Exp. 4)

Como puede observarse en la tabla, los resultados obtenidos por J48 son razonablemente buenos. En ilustración 7.19 aparecen recuadradas las instancias mal clasificadas.

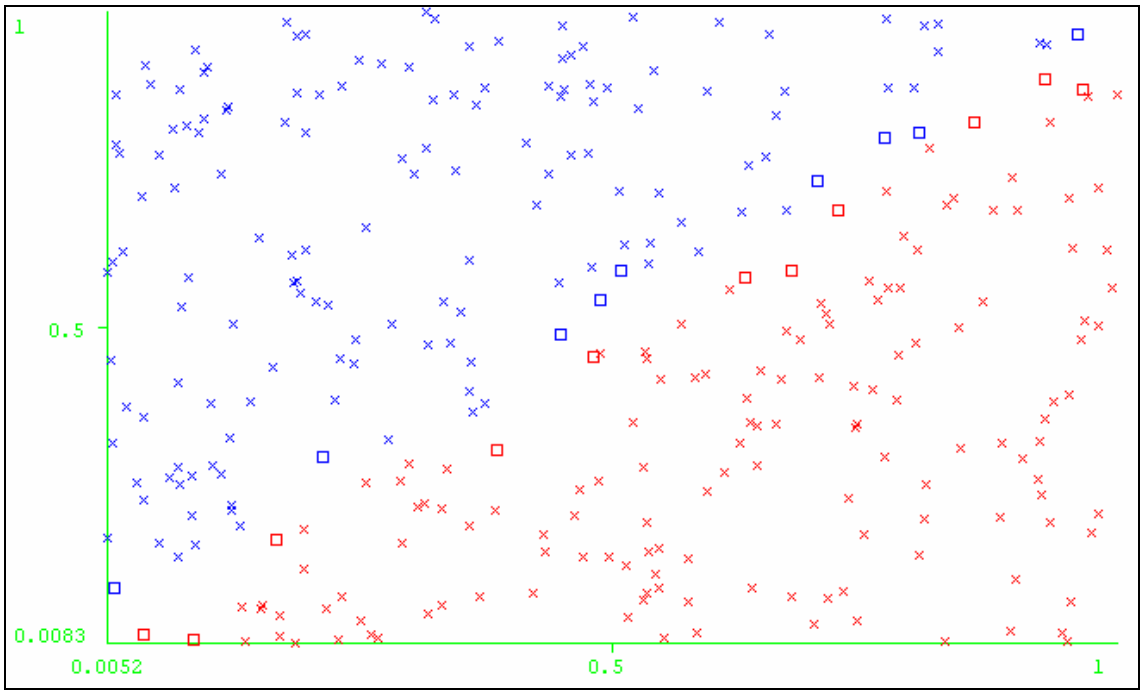

Ilustración 7.19 – Instancias mal clasificadas por J48 en aleatorios100.arff

Como puede observarse, las instancias mal clasificadas son aquellas que están en la frontera.

7.5.3 Experimentación con PMatrix

En este apartado se llevará a cabo la experimentación con el algoritmo desarrollado. En este experimento, al ser necesario trabajar con matrices de dos columnas, sólo se ejecutará con matrices completas. Si se trabajase con matrices simétricas o diagonales las matrices deben ser cuadradas, y esto implicaría que tuvieran cuatro columnas ya que el conjunto de datos tiene cuatro atributos.

De nuevo, se denominará como 4.X a cada uno de los sucesivos experimentos, con distintos parámetros, que se llevarán a cabo; siendo X un número natural que se irá incrementando gradualmente. Los valores de los parámetros se mostrarán en el formato tabular utilizado en los experimentos anteriores.

Por último, y cuando resulte beneficioso, se visualizarán los datos proyectados para observar cómo han sido proyectados con la mejor matriz encontrada.

Una vez explicado como será el proceso de experimentación con el algoritmo desarrollado se ejecutará el algoritmo con el primer tipo de individuo (matriz completa) y el resto de parámetros tendrán valores que garanticen una búsqueda lo más completa posible. En resumen, para el experimento 4.1 se utilizarán los siguientes valores:

Parámetro	Valor
Clasificador base	J48
Cruce	Activado
Constante de decremento	0.93
Exponente	1.0
Nº individuos en el cruce	8
Número columnas (atributos)	2
Nº generaciones sin mejora	7
Tamaño de la población	15
Prob. mutación individuo	65
Prob. mutación entrada (matriz)	20
Tamaño del torneo	6
Tipo Individuo	1

Tabla 7.18 – Valores de los parámetros de *PMatrix* en el Exp. 4.1

PMatrix clasifica correctamente al 51% de las instancias con la configuración de la tabla 7.18. Para el experimento 4.2 se utilizarán los siguientes valores:

Parámetro	Valor
Clasificador base	J48
Cruce	Activado
Constante de decremento	**0.96**
Exponente	1.0
Nº individuos en el cruce	**50**
Número columnas (atributos)	2
Nº generaciones sin mejora	**15**
Tamaño de la población	**100**
Prob. mutación individuo	65
Prob. mutación entrada (matriz)	20
Tamaño del torneo	**18**
Tipo Individuo	1

Tabla 7.19 – Valores de los parámetros de *PMatrix* en el Exp. 4.2

Con la configuración especificada en la tabla 7.19, *PMatrix* clasifica correctamente al 73.3333% de las instancias. En la ilustración 7.20 puede observarse en el fichero de traza del actual experimento cómo evoluciona el fitness del conjunto de datos original proyectado:

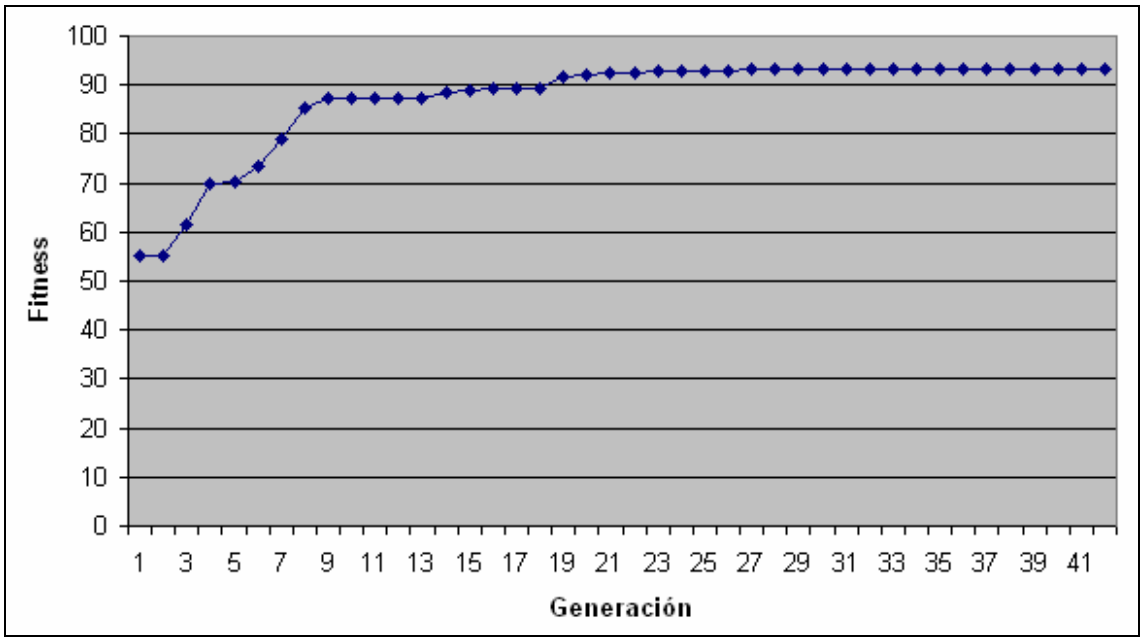

Ilustración 7.20 – Evolución del fitness de los datos proyectados (Exp 4.2).

En el experimento 4.3, se ejecutará el meta-algoritmo con el cruce desactivado, para ver si mejora el resultado evolucionando la población más que cruzando individuos seleccionados, asimismo, se probará con un exponente diferente. En resumen, para el experimento 4.3 se utilizarán los siguientes valores:

Parámetro	Valor
Clasificador base	J48
Cruce	**Desactivado**
Constante de decremento	0.96
Exponente	**2.0**
Nº individuos en el cruce	50
Número columnas (atributos)	2
Nº generaciones sin mejora	15
Tamaño de la población	100
Prob. mutación individuo	65
Prob. mutación entrada (matriz)	**100**
Tamaño del torneo	**45**
Tipo Individuo	1

Tabla 7.20 – Valores de los parámetros de *PMatrix* en el Exp. 4.3

Sin embargo, el experimento 4.3 sólo logra clasificar correctamente al 68.6667% de las instancias. En la ilustración 7.21 se muestra cómo quedan los datos proyectados:

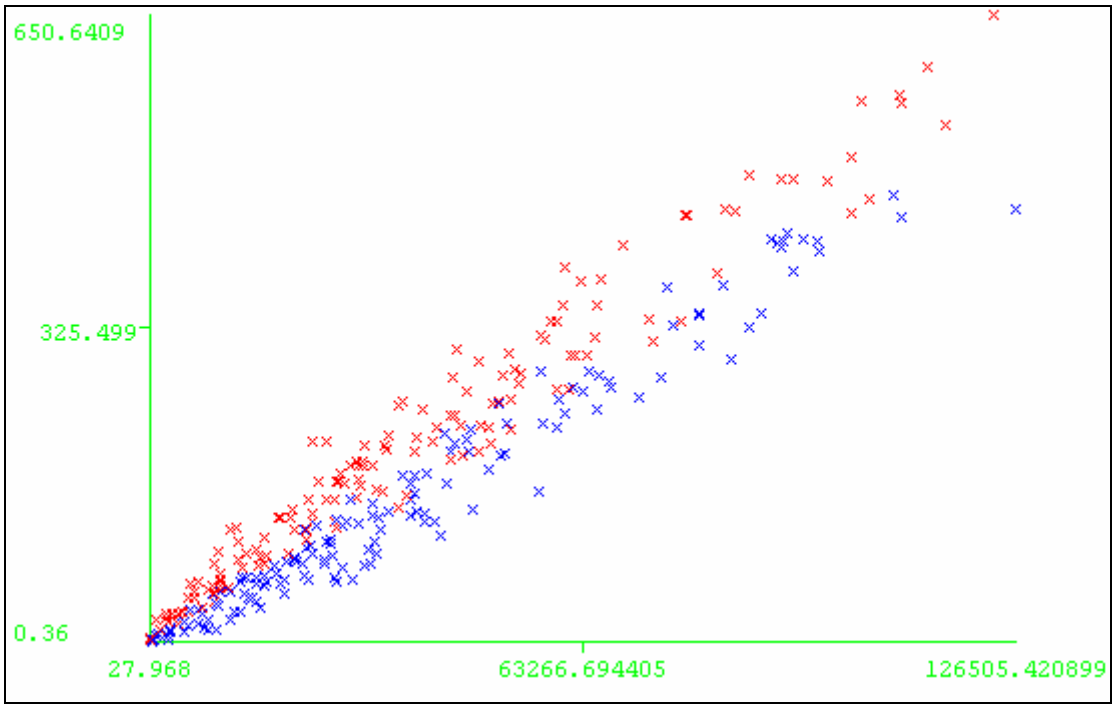

Ilustración 7.21 – Proyección del conjunto original en el experimento 4.3

Quizás el problema por el que no se puede mejorar la predicción del clasificador base es debido a que una matriz de dos columnas tiene que evolucionar mucho para que descarte dos de los atributos y transforme los otros dos. Por este motivo, en el experimento 4.4 se utilizará una matriz de cuatro columnas. Se utilizarán los valores de la tabla 7.21:

Parámetro	Valor
Clasificador base	J48
Cruce	**Activado**
Constante de decremento	**0.97**
Exponente	**1.0**
Nº individuos en el cruce	**30**
Número columnas (atributos)	**-1**
Nº generaciones sin mejora	**12**
Tamaño de la población	100
Prob. mutación individuo	**80**
Prob. mutación entrada (matriz)	**30**
Tamaño del torneo	**11**
Tipo Individuo	**3**

Tabla 7.21 – Valores de los parámetros de *PMatrix* en el Exp. 4.4

El experimento 4.4 logra igualar el resultado obtenido por J48. El resultado obtenido es del 93.3333% de instancias clasificadas correctamente. Posteriores experimentos, con matrices simétricas y completas de cuatro columnas no encontraron una mejor solución.

En la tabla 7.22 se muestra una comparativa con los resultados de los distintos experimentos:

Experimento	% Aciertos
Identidad[7]	93.3333%
Experimento 4.1	51%
Experimento 4.2	73.3333%
Experimento 4.3	68.6667%
Experimento 4.4	93.3333%

Tabla 7.22 – Comparativa de los resultados de los experimentos 4.1 a 4.4

7.5.4 Conclusiones

Aunque en este último experimento *PMatrix* no haya sido capaz de encontrar una matriz para proyectar el conjunto de datos original que sea mejor clasificada por *J48* que el conjunto de datos original sin proyectar, no significa que *PMatrix* no sea capaz de encontrarla; simplemente, en los experimentos realizados no ha sido encontrada. Por otro lado, es también posible que el incremento en el número de parámetros a aprender (los coeficientes de la matriz), hagan que el algoritmo sobreadapte, lo que explicaría los malos resultados de los experimentos 4.2 y 4.3.

En cuanto a los tiempos, siguen siendo ejecuciones bastante rápidas. Por ejemplo, el experimento 4.3 tardó cuatro minutos en un Pentium IV a 3.2 GHz con Windows XP y el experimento 4.4 fue llevado a cabo en menos de 100 segundos.

[7] Corresponde al primer experimento, realizado con J48 sin *PMatrix*.

7.6 Conclusiones globales

Los cuatro experimentos han sometido a *PMatrix* a una exhaustiva prueba de su correcto funcionamiento y su potencia. Salvo en el experimento 4, en todos los experimentos ha mejorado el porcentaje de aciertos conseguido sin *PMatrix*; siendo destacable las mejoras obtenidas en los dominios sintéticos *rectas45* y *rectas0*.

Además, los tiempos necesarios por *PMatrix* para llevarlos a cabo han sido muy pequeños, en relación con las grandes poblaciones con las que trabajaba y recordando que en todas ellas el modo de test elegido era la validación cruzada con 10 hojas.

En la tabla 7.23 se muestra una tabla resumen con los resultados en test de los dominios probados en los diferentes experimentos con todos los tipos de matriz y la matriz identidad (sin *PMatrix*):

Tipo de Matriz	% Aciertos	Dominio
Identidad	10.5%	*rectas45*
Completa	99%	
Simétrica	99.5%	
Diagonal	10.5%	
Identidad	9.5%	*rectas0*
Completa	-	
Simétrica	-	
Diagonal	100%	
Identidad	73.8281%	*Diabetes*
Completa	76.6927%	
Simétrica	-	
Diagonal	74.0885%	
Identidad	93.3333%	*aleatorios100*
Completa	72.6667%	
Simétrica	-	
Diagonal	93.3333%	

Tabla 7.23 – Comparativa de los resultados de todos los experimentos

8. Conclusiones y líneas futuras

En este capítulo se hará un breve repaso a las conclusiones sobre el trabajo desarrollado y qué mejoras tienen cabida en el mismo.

8.1 Conclusiones

En este proyecto se ha desarrollado *PMatrix* (matriz de proyecciones), un método que mejora a los algoritmos de aprendizaje automático transformando los datos iniciales. Dicha transformación se optimiza mediante técnicas evolutivas. El desarrollo del meta-algoritmo *PMatrix* ha supuesto disponer de un algoritmo que permita, lejos de las insuficientes proyecciones aleatorias, ampliar y reducir la dimensionalidad, en la medida que se desee, de los conjuntos de datos con que se trabaja.

En todos los casos, haya reducción o no de la dimensionalidad, el algoritmo busca utilizando computación evolutiva la mejor proyección para cada determinado clasificador base. Al poder elegir cualquier clasificador base integrado en Weka, pueden encontrarse proyecciones óptimas para clasificadores de todo tipo: lineales, de reglas… y, con el paso del tiempo, el número de clasificadores base seleccionables irá en aumento.

Es destacable la rapidez con que realiza la búsqueda de la mejor matriz y los resultados obtenidos, principalmente en los dominios sintéticos, algo que da buena prueba de su correcto y eficiente funcionamiento.

Además, a pesar de que la forma de funcionar del meta-algoritmo involucraba una gran complejidad para ser integrado en Weka, ha sido integrado sin haber modificado el núcleo de Weka, con ciertas suposiciones que han de tenerse en cuenta cuando sea utilizado.

Por último, permite una exhaustiva y completa configuración de los más importantes parámetros que intervienen en el algoritmo evolutivo.

8.2 Líneas futuras

A pesar de que el algoritmo cumple su objetivo de una manera relativamente eficiente (en cuanto a tiempo se refiere), esto no quiere decir que el algoritmo no pueda aceptar mejoras.

En primer lugar, aunque los conjuntos de datos utilizados en los experimentos no son pequeños (diabetes tiene 768 instancia y 9 atributos), podrían presentársele conjuntos de datos mucho mayores, del orden de decenas de miles de instancias y/o podría ser requerida un población mucho mayor que las utilizadas en los experimentos del capítulo 7, del orden de miles o decenas de miles de individuos. En estos casos, el tiempo necesario para la ejecución del meta-algoritmo podría aumentar drásticamente y dichos experimentos podrían llegar a ser intratables. Por tanto, para solucionar este problema podría paralelizarse el cómputo en diferentes máquinas.

Además, no siempre se conoce cuál será la dimensión óptima del espacio proyectado y, en muchos casos, ni siquiera se tiene una vaga idea sobre su número. A pesar de esto, uno de los parámetros que ha de configurarse en el meta-algoritmo es el número de columnas de la matriz, es decir, la dimensión del espacio proyectado. Una posible mejora sería ampliar, por ejemplo, con el número -2, el rango de valores posibles para dicho parámetro; permitiendo, cuando así esté configurado, que el algoritmo descubra cuál es la mejor dimensión, es decir, que determine si es beneficioso ampliar o reducir la dimensión y en qué medida.

Por último, podrían utilizarse técnicas evolutivas más complejas, tales como coevolución, algoritmos meméticos, evolución con nichos…, que podrían redundar en una mejora de los resultados.

ANEXOS

A. Detalles de implementación

En este anexo se incluyen los detalles de implementación, es decir, para cada una de las clases desarrolladas se detalla qué paquetes se han importado y qué métodos y atributos se han definido.

A modo de recordatorio, la ilustración A.1 muestra el diagrama de clases de las clases que componen el meta-algoritmo *PMatrix*.

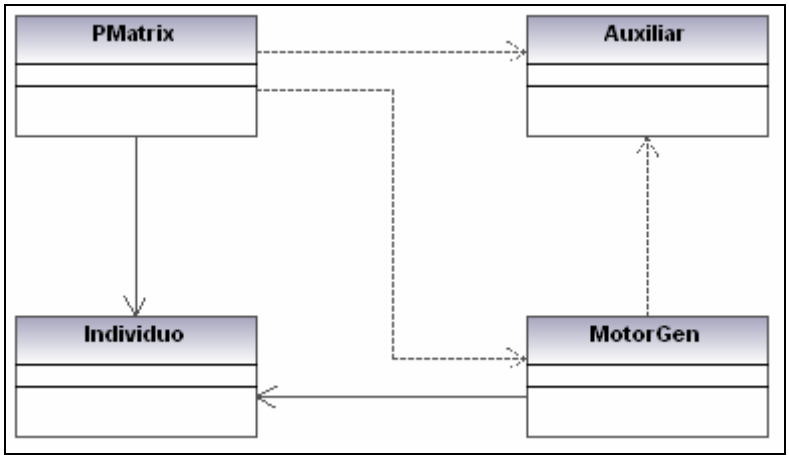

Ilustración A.1 – Diagrama de clases del meta-algoritmo *PMatrix*

A.1 Clase Individuo

En este apartado se detalla qué paquetes se han importado y qué métodos y atributos se han definido para la clase *Individuo*. El diagrama de clases completo de la clase *Individuo* se muestra en la ilustración A.2.

Ilustración A.2 – Diagrama de clases de la clase *Individuo*

A.1.1 Importación de paquetes

En la tabla A.1 se indican los paquetes importados en la clase *Individuo* y se describe brevemente qué contienen y/o por qué motivo son utilizados.

Paquete	Descripción
`java.text.DecimalFormat`	Útil para formatear la escritura de números reales.
`java.util.Random`	Permite generar números aleatorios.

Tabla A.1 – Importación de paquetes en la clase *Individuo*

A.1.2 Definición de atributos

En la tabla A.2 se listan los atributos definidos en la clase *Individuo* indicando su visibilidad, tipo y nombre. Asimismo, se realiza una breve descripción de cada uno de ellos.

Visibilidad	Tipo	Nombre	Descripción
Privado	*int*	*numAtributos*	Número de filas de la matriz, que es igual al número de atributos del conjunto de datos original.
Privado	*int*	*numColumnas*	Numero de columnas de la matriz, que determina la dimensión del conjunto de datos proyectado.
Privado	*double[][]*	*matrizPesos*	Matriz de pesos del individuo, con la que se realizara la proyección.
Privado	*int*	*tipo*	Tipo de individuo (matriz): 1 si completa, 2 si simétrica, 3 si diagonal.
Privado	*double*	*valorMin*	Valor mínimo de las entradas de la matriz de pesos.
Privado	*double*	*valorMax*	Valor máximo de las entradas de la matriz de pesos.
Privado	*double*	*diferencia*	Diferencia entre el valor máximo y mínimo de las entradas de la matriz.
Privado	*double*	*varianza*	Varianza del individuo.
Privado	*double*	*fitness*	Fitness o aptitud del individuo.
Privado	*boolean*	*evaluado*	Determina si el fitness o aptitud del individuo ha sido calculado.

Tabla A.2 – Atributos definidos en la clase *Individuo*

A.1.3 Definición de métodos

En este subapartado se listan los métodos implementados en la clase *Individuo* indicando su visibilidad y sus argumentos de entrada y de salida. Asimismo, se realiza una breve descripción de cada uno de ellos.

Los métodos accedentes y mutadores no serán detallados como el resto de métodos por motivos de espacio y legibilidad, no obstante pueden consultarse en la ilustración A.2.

Individuo()		
Descripción	Constructor parametrizado de individuo.	
Visibilidad	Público	
Argumentos de entrada	*int numAtributos*	Numero de filas de la matriz.
	int numColumnas	Numero de columnas de la matriz.
	int tipo	Tipo de individuo (matriz): 1 si completa, 2 si simétrica, 3 si diagonal.

Tabla A.3 – Método Individuo() de la clase Individuo

Individuo()		
Descripción	Constructor parametrizado de individuo.	
Visibilidad	Público	
Argumentos	*Individuo ind*	Individuo a ser copiado.

de entrada		

Tabla A.4 – Método Individuo() de la clase Individuo

toString()		
Descripción	Devuelve una cadena con información del individuo para mostrarla por pantalla o ser volcada a un fichero.	
Visibilidad	Público	
Argumento de salida	*String*	Cadena con información del individuo.

Tabla A.5 – Método toString() de la clase Individuo

matrizPesosToString()		
Descripción	Devuelve una cadena de texto con la matriz de pesos del individuo para ser impresa por pantalla o volcada a un fichero.	
Visibilidad	Público	
Argumento de salida	*String*	Cadena de texto con la matriz de pesos del individuo.

Tabla A.6 – Método matrizPesosToString() de la clase Individuo

matrizToString()		
Descripción	Devuelve una cadena de texto con la matriz de pesos pasada como argumento para ser impresa por pantalla o volcada a un fichero.	
Visibilidad	Público y estático	
Argumentos de entrada	*double[][] matriz*	Matriz pesos a ser impresa.
	int atribs	Número de filas de la matriz.
	int cols	Número de columnas de la matriz.
Argumento de salida	*String*	Cadena de texto con la matriz de pesos pasada como argumento.

Tabla A.7 – Método matrizToString() de la clase Individuo

incremEntrada()		
Descripción	Incrementa en inc (que puede ser positivo o negativo) una entrada de la matriz.	
Visibilidad	Público	
Argumentos de entrada	*int atrib*	Fila de la matriz cuya entrada se desea modificar.
	int inst	Columna de la matriz cuya entrada se desea modificar.
	double inc	Cantidad en que se desea incrementar dicha entrada.

Tabla A.8 – Método incremEntrada() de la clase Individuo

A.2 Clase Auxiliar

En este apartado se detalla qué paquetes se han importado y qué métodos y atributos se han definido para la clase *Auxiliar*. El diagrama de clases completo de la clase *Auxiliar* se muestra en la ilustración A.3.

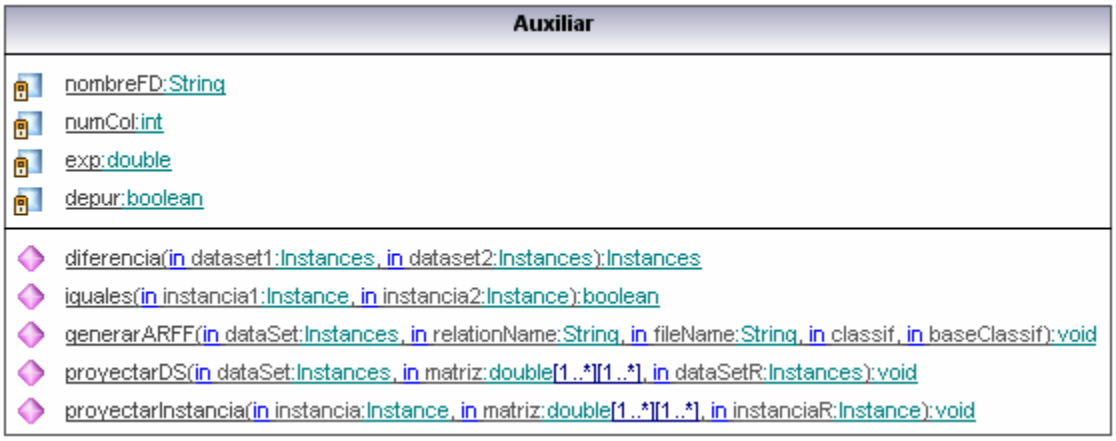

Ilustración A.3 – Diagrama de clases de la clase *Auxiliar*

A.2.1 Importación de paquetes

En la tabla A.9 se indican los paquetes importados en la clase Auxiliar y se describe brevemente qué contienen y/o el motivo de dicha importación.

Paquete	Descripción
`java.io.FileWriter`	Permite escribir caracteres en ficheros.
`java.io.IOException`	Permite el trabajo con excepciones de entrada/salida.
`java.io.PrintWriter`	Permite trabajar con flujos de texto.
`java.text.DecimalFormat`	Útil para formatear la escritura de números reales.
`javax.swing.JOptionPane`	Permite mostrar ventanas emergentes con mensajes de información, error…
`weka.core.Instance`	Clase que representa cada una de las instancias de un conjunto de datos.
`weka.core.Instances`	Clase que representa a un conjunto de datos.

Tabla A.9 – Importación de paquetes en la clase *Auxiliar*

A.2.2 Definición de atributos

En la tabla A.10 se listan los atributos definidos en la clase *Auxiliar* indicando su visibilidad, tipo y nombre. Asimismo, se realiza una breve descripción de cada uno de ellos.

Visibilidad	Tipo	Nombre	Descripción
Privado	*String*	*nombreFD*	Nombre, con ruta absoluta, del fichero de depuración.
Privado	*int*	*numCol*	Numero de columnas de la matriz de pesos.
Privado	*double*	*exp*	Exponente al cual elevar las entradas proyectadas del conjunto de datos destino.
Privado	*boolean*	*depur*	Indica si la depuración esta activada.

Tabla A.10 – Atributos definidos en la clase *Auxiliar*

A.2.3 Definición de métodos

En este subapartado se listan los métodos implementados en la clase *Auxiliar* indicando su visibilidad y sus argumentos de entrada y de salida. Asimismo, se realiza una breve descripción de cada uno de ellos.

Los métodos accedentes y mutadores no serán detallados como el resto de métodos por motivos de espacio y legibilidad, no obstante pueden consultarse en la ilustración A.3.

diferencia()		
Descripción	Calcula la diferencia entre dos conjuntos de datos.	
Visibilidad	Público	
Argumentos de entrada	*Instances dataset1*	Conjunto de datos (minuendo de la operación).
	Instances dataset2	Conjunto de datos (sustraendo de la operación).
Argumento de salida	*Instances*	Conjunto de datos con las instancia del conjunto de datos minuendo que no aparecen en el sustraendo.

Tabla A.11 – Método diferencia() de la clase Auxiliar

iguales()		
Descripción	Determina si dos instancias son iguales, no solo por los valores de sus atributos, sino hasta por el conjunto de datos al que pertenecen.	
Visibilidad	Público	
Argumentos de entrada	*Instance instancia1*	Primera de las instancias a comparar.
	Instance instancia2	Segunda de las instancias a comparar.
Argumento de salida	*boolean*	Indica si las instancias pasadas como argumento son iguales.

Tabla A.12 – Método iguales() de la clase Auxiliar

generarARFF()	
Descripción	Genera un fichero ARFF valido conteniendo el conjunto de datos pasado como argumento. Además, aparece comentado el nombre del clasificador que lo genere y el nombre del clasificador base que ha

	utilizado.	
Visibilidad	Público	
Argumentos de entrada	*Instantes dataSet*	Conjunto de datos a ser generado.
	String relationName	Nombre de la relación.
	String fileName	Nombre (con dirección absoluta) del fichero ARFF.
	String classif	Nombre del clasificador (será *PMatrix*).
	String baseClassif	Nombre del clasificador base.

Tabla A.13 – Método generarARFF() de la clase Auxiliar

proyectarDS()		
Descripción	Proyecta un conjunto de datos con una determinada matriz de pesos.	
Visibilidad	Público	
Argumentos de entrada	*Instances dataSet*	Conjunto de datos a proyectar.
	double[][] matriz	Matriz de pesos con la cual realizar la proyección.
	Instances dataSetR	Conjunto de datos proyectado.

Tabla A.14 – Método proyectarDS() de la clase Auxiliar

proyectarInstancia()		
Descripción	Proyecta una instancia de datos con una determinada matriz de pesos.	
Visibilidad	Público	
Argumentos de entrada	*Instance instancia*	Instancia a proyectar.
	double[][] matriz	Matriz de pesos con la cual realizar la proyección.
	Instance instanciaR	Instancia proyectada.

Tabla A.15 – Método proyectarInstancia() de la clase Auxiliar

A.3 Clase MotorGen

En este apartado se detalla qué paquetes se han importado y qué métodos y atributos se han definido para la clase *MotorGen*. El diagrama de clases completo de la clase *MotorGen* se muestra en la ilustración A.4.

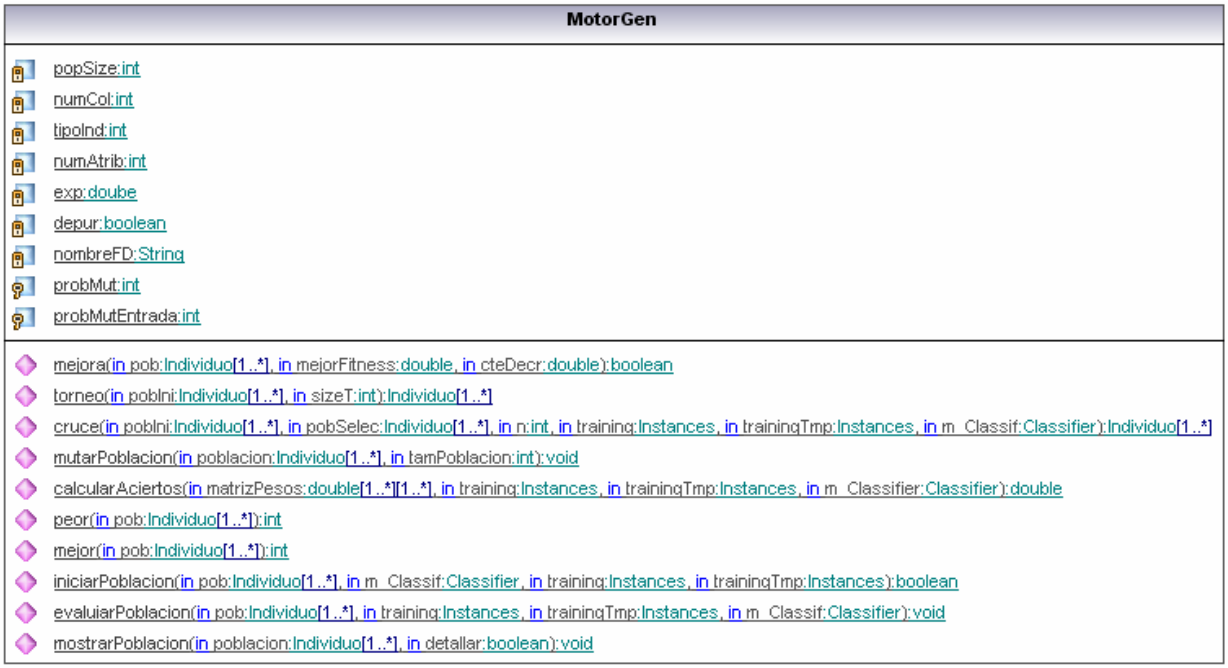

Ilustración A.4 – Diagrama de clases de la clase *MotorGen*

A.3.1 Importación de paquetes

En la tabla A.16 se indican los paquetes importados en la clase *MotorGen* y se describe brevemente qué contienen y/o por qué motivo son utilizados.

Paquete	Descripción
`java.io.FileWriter`	Permite escribir caracteres en ficheros.
`java.io.IOException`	Permite el trabajo con excepciones de entrada/salida.
`java.io.PrintWriter`	Permite trabajar con flujos de texto.
`java.text.DecimalFormat`	Útil para formatear la escritura de números reales.
`java.util.Random`	Permite generar números aleatorios.
`javax.swing.JOptionPane`	Permite mostrar ventanas emergentes con mensajes de información, error…
`weka.classifiers.Classifier`	Clase abstracta de la que heredan todos los clasificadores en Weka.
`weka.classifiers.Evaluation`	Clase que permite evaluar los conjuntos de datos con los modelos construidos con diferentes modos de test.
`weka.core.Instances`	Clase que representa a un conjunto de datos.

Tabla A.16 – Importación de paquetes en la clase *MotorGen*

A.3.2 Definición de atributos

En la tabla A.17 se listan los atributos definidos en la clase *MotorGen* indicando su visibilidad, tipo y nombre. Asimismo, se realiza una breve descripción de cada uno de ellos.

Visibilidad	Tipo	Nombre	Descripción
Privado	*int*	*popSize*	Tamaño de la población
Privado	*int*	*numCol*	Número de columnas de la matriz de pesos
Privado	*int*	*tipoInd*	Tipo de matriz: 1 si completa, 2 si simétrica, 3 si diagonal
Privado	*int*	*numAtrib*	Número de filas de la matriz de pesos, coincide con el numero de atributos del espacio origen
Privado	*double*	*exp*	Exponente al cual se eleva el espacio de datos tras la proyección
Privado	*boolean*	*depur*	Indica si la depuración esta activada
Privado	*int*	*probMutEntrada*	Probabilidad de mutación de cada entrada de la matriz
Privado	*int*	*probMut*	Probabilidad de mutación de cada individuo (matriz)
Privado	*String*	*nombreFD*	Nombre (con ruta absoluta) del fichero de depuración.

Tabla A.17 – Atributos definidos en la clase *MotorGen*

A.3.3 Definición de métodos

En este subapartado se listan los métodos implementados en la clase *MotorGen* indicando su visibilidad y sus argumentos de entrada y de salida. Asimismo, se realiza una breve descripción de cada uno de ellos.

Los métodos accedentes y mutadores no serán detallados como el resto de métodos por motivos de espacio y legibilidad, no obstante pueden consultarse en la ilustración A.4.

mejora()		
Descripción	Devuelve verdadero si ha habido una mejora, en caso contrario devuelve falso y decrementa el valor de las varianzas de los individuos de la población.	
Visibilidad	Público	
Argumentos de entrada	*Individuo[] pob*	Población de individuos a comprobar si mejoran el mejor fitness hallado hasta el

		momento.
	double mejorFitness	Mejor fitness hallado hasta el momento.
	double cteDecr	Constante de decremento de las varianzas de los individuos.
Argumento de salida	*double*	*Double.NaN* si no mejora el fitness; en caso contrario, el nuevo mejor fitness

Tabla A.18 – Método mejora() de la clase MotorGen

torneo()		
Descripción	Selecciona una población de los mejores individuos con la técnica de torneos.	
Visibilidad	Público	
Argumentos de entrada	*Individuo[] pobIni*	Población de individuos.
	int sizeT	Tamaño de los torneos.
Argumento de salida	*Individuo[]*	Población con los individuos seleccionados.

Tabla A.19 – Método torneo() de la clase MotorGen

cruce()		
Descripción	Implementa el método de cruce, es decir, genera n nuevos individuos, los evalúa y los inserta en la población reemplazándolos, uno tras otro, por el peor individuo que hubiera en la población en ese momento.	
Visibilidad	Público	
Argumentos de entrada	*Individuo[] pobIni*	Población de individuos.
	Individuo[] pobSelec	Población seleccionada de individuos (torneos).
	int n	Numero de nuevos individuos a crear.
	Instances training	Conjunto de datos de entrenamiento.
	Instances trainingTmp	Conjunto de datos temporal, será utilizado para almacenar las diferentes proyecciones.
	Classifier m_Classif	Clasificador base con el que evaluar a los nuevos individuos.
Argumento de salida	*Individuo[]*	Población tras el cruce y reemplazamiento de los individuos viejos por los nuevos.

Tabla A.20 – Método cruce() de la clase MotorGen

mutarPoblacion()		
Descripción	Muta la población, mutando para ello con probabilidad probMut a cada individuo. Asimismo, el individuo a ser mutado lo será por la mutación de cada una de las entradas de su matriz con probabilidad probMutEntry.	
Visibilidad	Público	
Argumentos de entrada	*Individuo[] poblacion*	Población de individuos a ser mutada.

	int tamPoblacion	Tamaño de la población de individuos a ser mutada.

Tabla A.21 – Método mutarPoblacion() de la clase MotorGen

calcularAciertos()		
Descripción	Evalúa un conjunto de datos proyectado con un determinado clasificador utilizando validación cruzada de 10 hojas como modo de test. Devuelve el porcentaje de aciertos si la clase es nominal y el error cuadrático medio si la clase es numérica.	
Visibilidad	Público	
Argumentos de entrada	*double[][] matrizPesos*	Matriz de pesos con la que realizar la proyección.
	Instances training	Conjunto de datos original.
	Instances trainingTmp	Conjunto de datos temporal, contendrá el conjunto de datos proyectado.
	Classifier m_Classif	Clasificador con el que construir el modelo y evaluar el conjunto de datos.
Argumento de salida	*double*	Respecto al conjunto de datos proyectados, porcentaje de aciertos si es nominal y error cuadrático medio si es numérica, es decir, el fitness de un nuevo individuo.

Tabla A.22 – Método calcularAciertos() de la clase MotorGen

mejor()		
Descripción	Devuelve la posición del individuo con mejor fitness de una población.	
Visibilidad	Público	
Argumentos de entrada	*Individuo[] pob*	Población de individuos donde buscar el mejor fitness.
Argumento de salida	*int*	Posición del individuo con mejor fitness.

Tabla A.23 – Método mejor() de la clase MotorGen

peor()		
Descripción	Devuelve la posición del individuo con peor fitness de una población.	
Visibilidad	Público	
Argumentos de entrada	*Individuo[] pob*	Población de individuos donde buscar el peor fitness.
Argumento de salida	*int*	Posición del individuo con menor fitness.

Tabla A.24 – Método peor() de la clase MotorGen

iniciarPoblacion()

Descripción	Crea una población de individuos con un tamaño determinado.	
Visibilidad	Público	
Argumentos de entrada	*Individuo[] pob*	Población de individuos a crear.
	Classifier m_Classif	Clasificador con el que evaluar el fitness de los nuevos individuos.
	Instances training	Conjunto de datos original.
	Instances trainingTmp	Conjunto de datos que contendrá el espacio proyectado.
Argumento de salida	*boolean*	Booleano que indica si ha habido algún fallo en la inicialización.

Tabla A.25 – Método iniciarPoblacion() de la clase MotorGen

evaluarPoblacion()		
Descripción	Evalúa los individuos no evaluados de una población de individuos.	
Visibilidad	Público	
Argumentos de entrada	*Individuo[] pob*	Población de individuos a evaluar.
	Instances training	Conjunto de datos original.
	Instances trainingTmp	Conjunto de datos que contendrá el espacio proyectado.
	Classifier m_Classif	Clasificador con el que evaluar el fitness de los nuevos individuos.

Tabla A.26 – Método evaluarPoblacion() de la clase MotorGen

mostrarPoblacion()		
Descripción	Redirige al fichero de depuración la impresión de la población de individuos.	
Visibilidad	Público	
Argumentos de entrada	*Individuo[] población*	Población a mostrar.
	boolean detallar	Si se requiere detalle, se muestra adicionalmente la matriz de pesos y la varianza del individuo.

Tabla A.27 – Método mostrarPoblacion() de la clase MotorGen

A.4 Clase PMatrix

En este apartado se detalla qué paquetes se han importado y qué métodos y atributos se han definido para la clase *PMatrix*. El diagrama de clases completo de la clase *PMatrix* se muestra en la ilustración A.5.

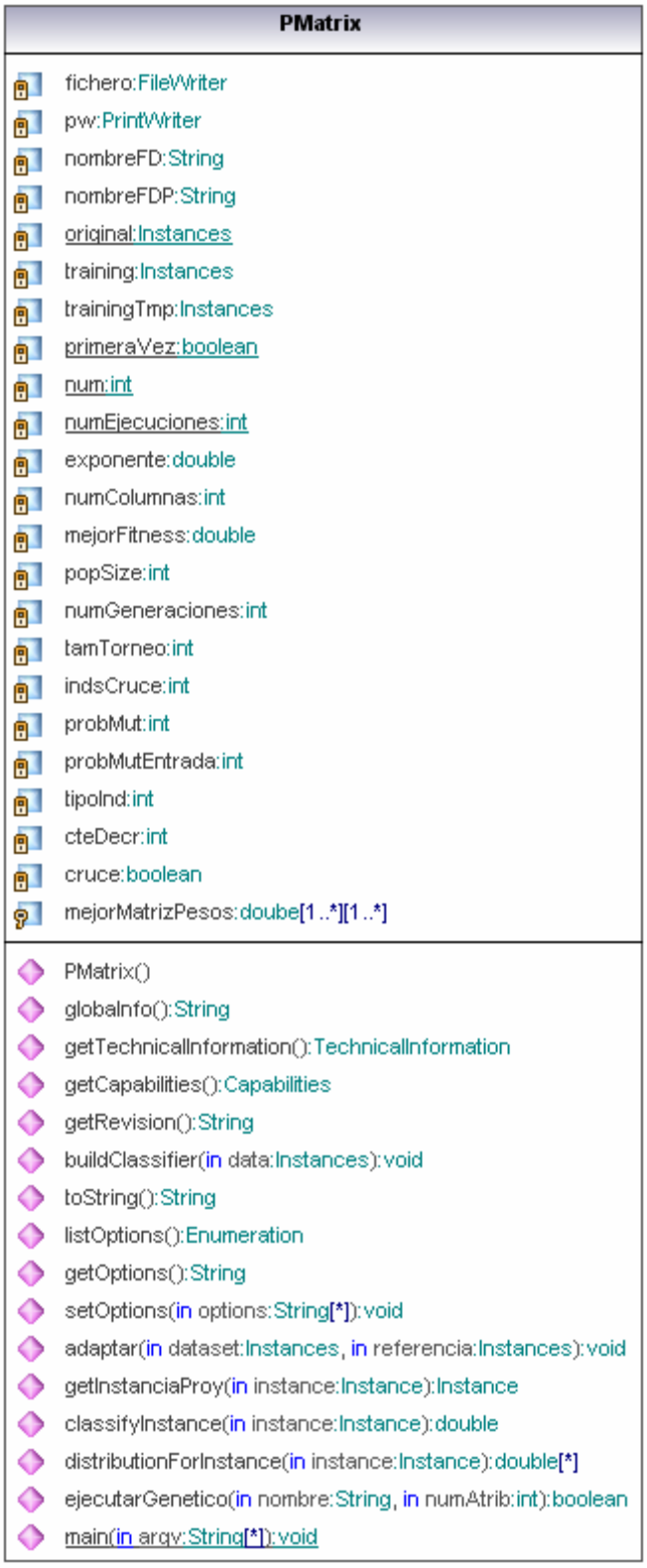

Ilustración A.5 – Diagrama de clases de la clase *PMatrix*

A.4.1 Importación de paquetes

En la tabla A.28 se indican los paquetes importados en la clase *PMatrix* y se describe brevemente qué contienen y/o por qué motivo son utilizados.

Paquete	Descripción
`java.text.DecimalFormat`	Útil para formatear la escritura de números reales.
`java.util.Random`	Permite generar números aleatorios.
`java.io.*`	Permite trabajar con entrada/salida.
`java.util.Date`	Permite trabajar con fechas.
`java.util.Enumeration`	Permite generar una serie de elementos, por ejemplo, para recorrer un vector.
`java.util.Vector`	Permite trabajar con vectores.
`javax.swing.JOptionPane`	Permite mostrar ventanas emergentes con mensajes de información, error…
`weka.classifiers`	Clase abstracta que contiene los clasificadores de Weka y, en concreto, *SingleClassifierEnhancer*, del cual hereda *PMatrix*.
`weka.classifiers.meta.pmatrix.*`	PMatrix necesita el resto de clases (*Auxiliar*, *MotorGen* e *Individuo*) para llevar a cabo su función.
`weka.core.*`	Núcleo de Weka, donde se implementan los objetos atributo, instancia, conjunto de datos…
`java.text.DecimalFormat`	Útil para formatear la escritura de números reales.

Tabla A.28 – Importación de paquetes en la clase *PMatrix*

A.4.2 Definición de atributos

En la tabla A.29 se listan los atributos definidos en la clase *PMatrix* indicando su visibilidad, tipo y nombre. Asimismo, se realiza una breve descripción de cada uno de ellos.

Visibilidad	Tipo	Nombre	Descripción
Privado	*FileWriter*	*fichero*	*FileWriter* para escribir en ficheros
Privado	*PrintWriter*	*pw*	*PrintWriter* para trabajar con flujos de texto
Privado	*String*	*nombreFD*	Nombre (con ruta absoluta) del fichero de depuración.
Privado	*String*	*nombreFDP*	Nombre (con ruta absoluta) del fichero de depuración de las proyecciones.
Privado	*Instances*	*original*	Conjunto de datos completo.

Privado	*Instances*	*training*	Conjunto de datos de entrenamiento
Privado	*Instances*	*trainingTmp*	Conjunto de datos de entrenamiento temporal; contendrá el conjunto de datos proyectado
Privado	*boolean*	*primeraVez*	Indica si es la primera vez que se invoca *buildClassifier*() con un conjunto de entrenamiento en la ejecución actual
Privado	*int*	*num*	Numero de veces que se ha invocado *buildClassifier*() con un conjunto de entrenamiento
Privado	*int*	*numEjecuciones*	Numero de ejecuciones del meta-algoritmo *PMatrix*
Privado	*double*	*exponente*	Exponente al cual se elevaran los valores del conjunto de datos tras la proyección
Privado	*int*	*numColumnas*	Numero de columnas de la matriz de pesos
Privado	*double*	*mejorFitness*	Mejor fitness encontrado hasta el momento
Privado	*int*	*popSize*	Tamaño de la población.
Privado	*int*	*numGeneraciones*	Numero de generaciones que el algoritmo seguirá buscando solución sin que haya habido mejoras.
Privado	*int*	*tamTorneo*	Tamaño de los torneos para la selección de individuos para su cruce.
Privado	*int*	*indsCruce*	Numero de nuevos individuos que se generaran en el cruce cada generación.
Privado	*int*	*probMut*	Probabilidad de mutación de cada individuo.
Privado	*int*	*probMutEntrada*	Probabilidad de mutación de cada una de las entradas de la matriz de pesos del individuo.
Privado	*int*	*tipo*	Tipo de individuo (matriz): 1 si completa, 2 si simétrica, 3 si completa.
Privado	*double*	*cteDecr*	Constante de decremento de las varianzas de los individuos.
Privado	*boolean*	*cruce*	Indica si esta activado el cruce de los individuos de la población.
Privado	*double[][]*	*mejorMatrizPesos*	Matriz de pesos con la que se ha obtenido mejor fitness.

Tabla A.29 – Atributos definidos en la clase *PMatrix*

A.4.3 Definición de métodos

En este subapartado se listan los métodos implementados en la clase *PMatrix* indicando su visibilidad y sus argumentos de entrada y de salida. Asimismo, se realiza una breve descripción de cada uno de ellos.

Los métodos accedentes y mutadores no serán detallados como el resto de métodos por motivos de espacio y legibilidad, no obstante pueden consultarse en la ilustración A.5.

PMatrix()		
Descripción	Constructor del meta-algoritmo.	
Visibilidad	Público	

Tabla A.30 – Método PMatrix() de la clase PMatrix

globalInfo()		
Descripción	Información general del meta-algoritmo. Será mostrada por Weka cuando muestre información de *PMatrix*.	
Visibilidad	Público	
Argumento de salida	*String*	Cadena de texto con dicha información global.

Tabla A.31 – Método globalInfo() de la clase PMatrix

getTechnicalInformation()		
Descripción	Información técnica del meta-algoritmo desarrollado. Tal información incluye el autor, el idioma, el año de construcción...	
Visibilidad	Público	
Argumento de salida	*TechnicalInformation*	Objeto con dicha informacion tecnica.

Tabla A.32 – Método getTechnicalInformation() de la clase PMatrix

getCapabilities()		
Descripción	Determina con que tipo de conjuntos de datos puede trabajar el meta-algoritmo desarrollado. En general, con clases nominales y numéricas y atributos numéricos, siempre que el clasificador base también pueda trabajar con ellas.	
Visibilidad	Público	
Argumento de salida	*Capabilities*	Objeto con las capacidades del individuo.

Tabla A.33 – Método getCapabilities() de la clase PMatrix

getRevision()		
Descripción	Devuelve una cadena de texto con la revisión.	
Visibilidad	Público	
Argumento de salida	*String*	Cadena con la revisión.

Tabla A.34 – Método getRevision() de la clase PMatrix

buildClassifier()		
Descripción	Construye un modelo para el conjunto de datos de entrenamiento pasado como argumento. Para ello, busca con computación evolutiva la matriz que mejor proyecta dicho conjunto de datos. Es el método central del meta-algoritmo.	
Visibilidad	Público	
Excepciones	Excepción que se lanza si el modelo no ha podido ser generado correctamente.	
Argumentos de entrada	*Instantes data*	Conjunto de datos de entrenamiento con que construir el modelo.

Tabla A.35 – Método buildClassifier() de la clase PMatrix

toString()		
Descripción	Devuelve información del meta-algoritmo, en concreto, una cadena de texto conteniendo el modelo generado.	
Visibilidad	Público	
Argumento de salida	*String*	Cadena de texto con la descripcion del modelo.

Tabla A.36 – Método toString() de la clase PMatrix

setOptions()		
Descripción	Asigna a los parámetros de configuración del meta-algoritmo los valores especificados por el usuario.	
Visibilidad	Público	
Excepciones	Excepción si los valores son incorrectos.	
Argumentos de entrada	*String[] options*	Valores de los parámetros de configuración.

Tabla A.37 – Método setOptions() de la clase PMatrix

getOptions()		
Descripción	Construye y devuelve un array con información textual sobre los parámetros de configuración y sus valores actuales.	
Visibilidad	Público	
Argumento de salida	*String*	String conteniendo las opciones y sus valores.

Tabla A.38 – Método getOptions() de la clase PMatrix

listOptions()		
Descripción	Lista las opciones disponibles para configurar.	
Visibilidad	Público	
Argumento de salida	*Enumeration*	Lista con dichas opciones.

Tabla A.39 – Método listOptions() de la clase PMatrix

adaptar()		
Descripción	Adapta el primer conjunto de datos pasado como argumento teniendo como referencia al segundo conjunto de datos pasado como argumento. El objetivo es modificar el primer conjunto de datos pasado como argumento para dejarlo preparado para contener posteriormente la proyección; esto implica que la clase sea el ultimo atributo y que tenga la dimensionalidad requerida.	
Visibilidad	Público	
Argumentos de entrada	*Instances dataSet*	Conjunto de datos a adaptar.
	Instances referencia	Conjunto de datos a tomar como referencia.

Tabla A.40 – Método adaptar() de la clase PMatrix

getInstanciaProy()		
Descripción	Proyectado la instancia pasada como argumento con la mejor matriz de pesos.	
Visibilidad	Público	
Argumentos de entrada	*Instance instance*	Instancia a proyectar.
Argumento de salida	*Instance*	Instancia proyectada.

Tabla A.41 – Método getInstanciaProy() de la clase PMatrix

classifyInstance()		
Descripción	Clasifica la instancia como argumento en el modelo recién construido.	
Visibilidad	Público	
Argumentos de entrada	*Instance instance*	Instancia a clasificar.
Argumento de salida	*double*	La clase de pertenencia mas probable para dicha instancia.

Tabla A.42 – Método classifyInstance() de la clase PMatrix

distributionForInstance()		
Descripción	Clasifica la instancia como argumento en el modelo recién construido.	
Visibilidad	Público	
Argumentos de entrada	*Instance instance*	Instancia a clasificar.
Argumento de salida	*double[]*	Array conteniendo la probabilidad de pertenencia de dicha instancia a cada una de las clases.

Tabla A.43 – Método distributionForInstance() de la clase PMatrix

ejecutarGenetico()	
Descripción	Lleva a cabo el bucle del algoritmo evolutivo. Inicializa los atributos de las clases auxiliares, inicializa la población y la evoluciona hasta que no mejora durante un cierto número de generaciones. Asimismo,

	escribe la traza de dicha evolución en el fichero de depuración.	
Visibilidad	Público	
Argumentos de entrada	*String nombre*	Nombre descriptivo del conjunto de datos (de entrenamiento) utilizado para buscar la matriz que mejor lo proyecte.
	int numAtrib	Numero de atributos del conjunto de datos.
Argumento de salida	*boolean*	Booleano que indica si la ejecución ha acabado correctamente o de lo contrario, no pudo iniciarse la población correctamente.

Tabla A.44 – Método ejecutarGenetico() de la clase PMatrix

B. Manual de Usuario

Este anexo contendrá el manual de usuario del software desarrollado. En primer lugar se realizará una breve descripción del software y, posteriormente, su instalación/ejecución y los requisitos software y hardware para la misma. El tercer apartado, el más extenso del manual, contendrá una guía de uso en la que se detallará su utilización desde el explorador y desde consola, asimismo se explicará cómo configurar el algoritmo modificando sus numerosos parámetros y se indicará cómo puede accederse a la ayuda del meta-algoritmo. A continuación, se detalla un ejemplo completo de uso, que corresponde con el primer experimento realizado en el capítulo 7 de la presente memoria. En última lugar, se detalla la información técnica del producto.

Puesto que el meta-algoritmo ha sido incluido en la versión de Weka para Windows, el manual de usuario enseñará a utilizar el meta-algoritmo en dicho sistema operativo.

B.1 Introducción

El meta-algoritmo desarrollado e integrado en Weka, *PMatrix*, tiene como objetivo buscar, haciendo uso de computación evolutiva, la matriz que mejor proyecte a un conjunto de datos evaluando dichas proyecciones con un determinado clasificador base. Dicha matriz, a elegir por el usuario, puede ser completa, simétrica o diagonal.

El usuario puede configurar numerosos parámetros del meta-algoritmo, tales como su clasificador base, el tamaño de la población, la probabilidad de mutación de cada individuo, tamaño de los torneos…

Además, debe funcionar correctamente sin importar el modo de test elegido y sus posibles parámetros.

Por último, *PMatrix* genera un directorio por cada ejecución conteniendo un fichero de traza y dos subdirectorios, *arff* y *models*, que contienen los ficheros *arff* de los conjuntos de training y test proyectados y los modelos que construye el algoritmo base para los conjuntos de training y test proyectados. El usuario puede activar la depuración, genera dos ficheros de texto adicionales: *debug* y *debugP*, conteniendo depuración del proceso de búsqueda y de las proyecciones de conjuntos de datos e instancias, respectivamente.

Este meta-algoritmo está integrado en Weka y ha sido desarrollado bajo licencia GPL[1], lo que significa que puede distribuirse y ser utilizado libremente.

La nueva versión ejecutable de Weka con *PMatrix* integrado debe funcionar obligatoriamente en Windows, no teniendo porqué funcionar en el resto de plataformas/sistemas operativos.

[1] GNU General Public License. http://www.gnu.org/copyleft/gpl.html

B.2 Instalación y requisitos

El meta-algoritmo desarrollado ha sido incluido en Weka y compilado junto con él, generando un fichero con extensión *jar* que contiene la versión 3.5.8 de Weka con *PMatrix*.

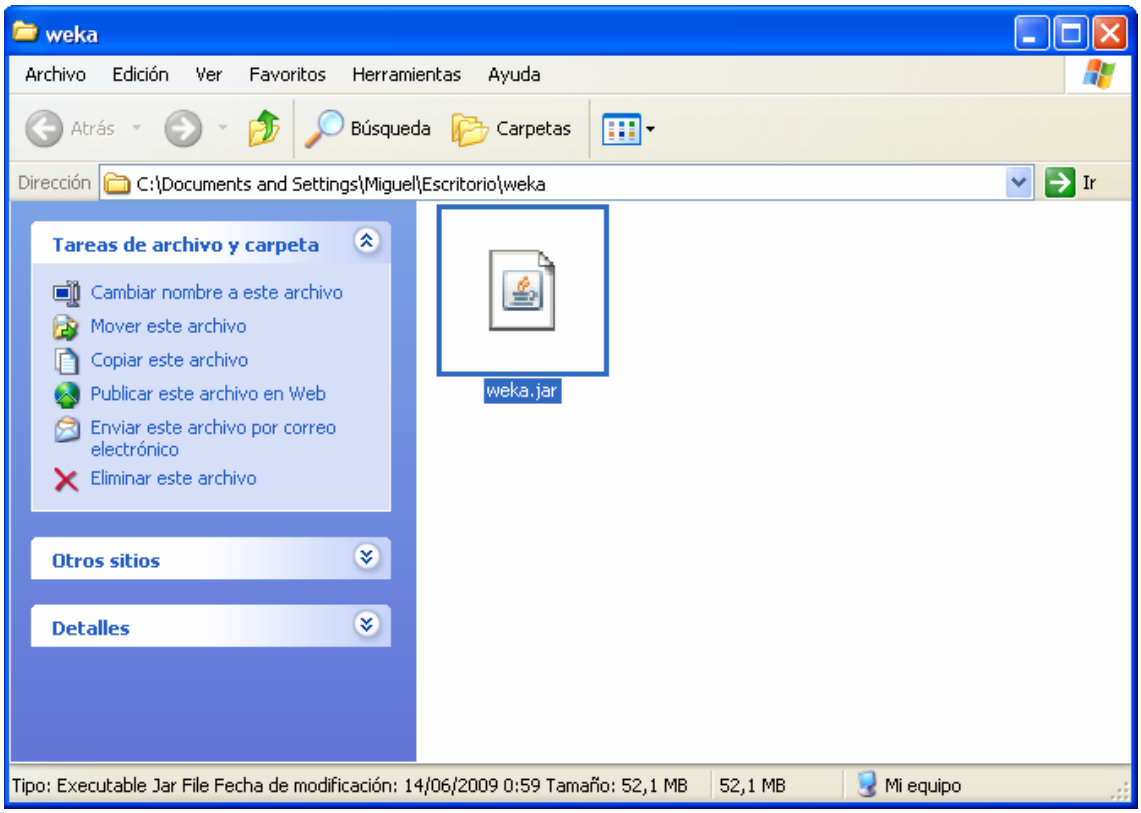

Ilustración B.1 – Fichero ejecutable de *Weka*

Al ser un fichero ejecutable no se requiere instalación, sino simplemente su ejecución. Puede ejecutarse haciendo doble clic sobre el icono o bien con el siguiente comando desde consola[2]:

[2] La consola de Windows puede abrirse desde el menú Inicio/Ejecutar con el comando *cmd*.

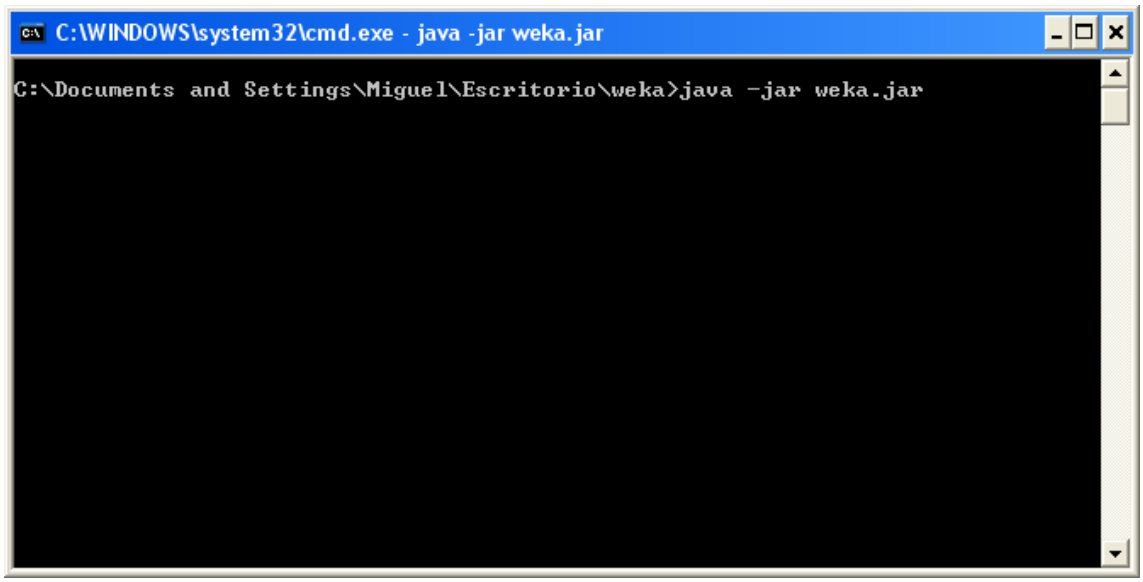

Ilustración B.2 – Ejecución de *Weka* desde consola

Una vez ejecutado, aparece la siguiente ventana desde la que, como se explicó en el capítulo 5, puede accederse al explorador y a la consola de Weka desde la pestaña *Applications*.

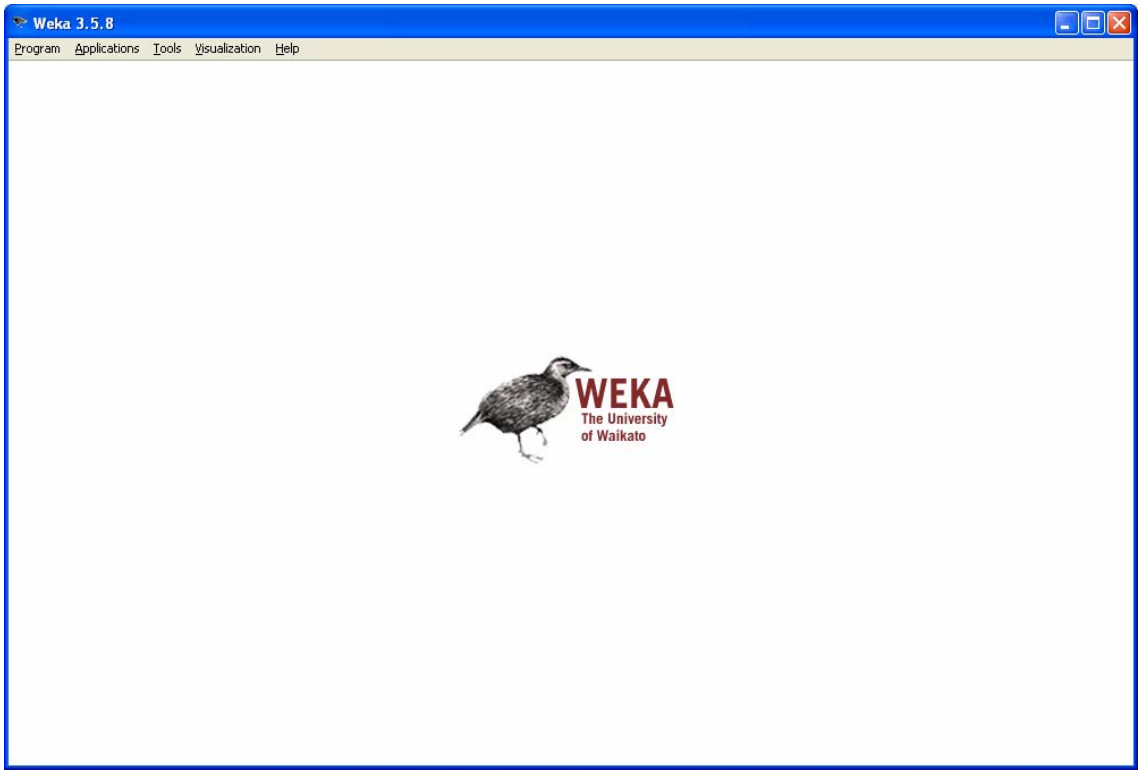

Ilustración B.3 – Ventana principal de *Weka*

Para ejecutar Weka es necesario Java 1.4 o superior. Puesto que el programa no ocupa demasiado espacio su ejecución es suficiente incluso con pequeñas cantidad de memoria y espacio, sin embargo, algunos algoritmos en funcionamiento requieren grandes cantidades de memoria, aproximadamente 30 MB de promedio aunque en ocasiones es

necesaria una cantidad mayor que ha de reservarse explícitamente con el siguiente comando:

```
java –Xms<mínima-memoria-asignada>M -Xmx<máxima-memoria-asignada>M -jar weka.jar
```

Donde –Xms y –Xmx indican, respectivamente, las cantidades mínima y máxima, expresadas en magabytes, de memoria RAM que se desean asignar.

B.3 Guía de uso

Este apartado contiene una extensa guía de uso en la que se detalla la utilización de *PMatrix* en Weka desde el explorador y desde consola, asimismo se explica cómo configurar el algoritmo modificando sus numerosos parámetros y se indica cómo acceder a la ayuda que proporciona el meta-algoritmo.

Antes de comenzar, conviene recordar que *PMatrix* sólo puede trabajar con clases nominales y numéricas y atributos numéricos.

B.3.1 Uso desde el explorador

Una vez en la ventana del explorador y con el conjunto de datos a utilizar cargado en Weka, para poder aprender con *PMatrix* debe seleccionarse desde la pestaña *Classify*.

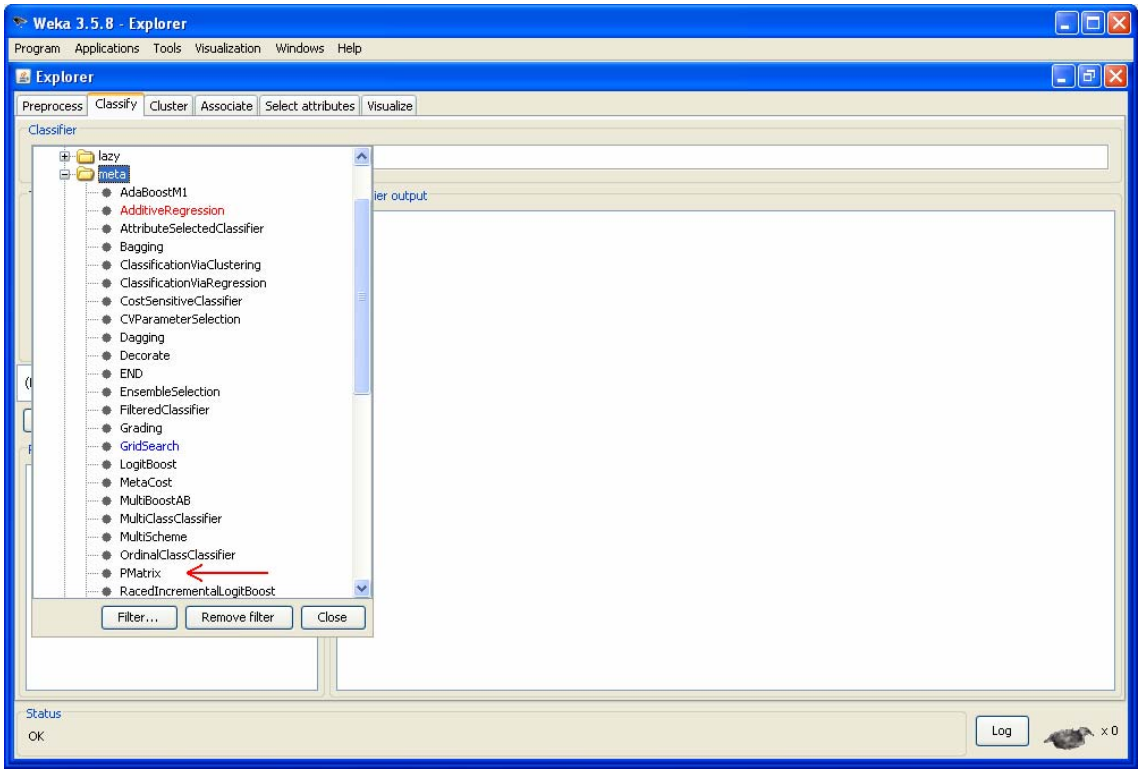

Ilustración B.4 – Pestaña *Classify* del explorador de *Weka*

Como se muestra en la ilustración B.4, *PMatrix* aparece, como el resto de meta-algoritmos, en orden alfabético en la carpeta *meta*. Una vez seleccionado *PMatrix* como el algoritmo del cual se quiere aprender, se configurarían sus parámetros[3] y se pulsaría el botón *Start*.

Los parámetros que pueden configurarse aparecen en la ilustración B.5. Cualquier configuración de parámetros puede ser guardada con el botón *Save* y posteriormente restaurada con *Open*.

[3] Los parámetros de *PMatrix* se explican con detalle en el apartado B.3.3.

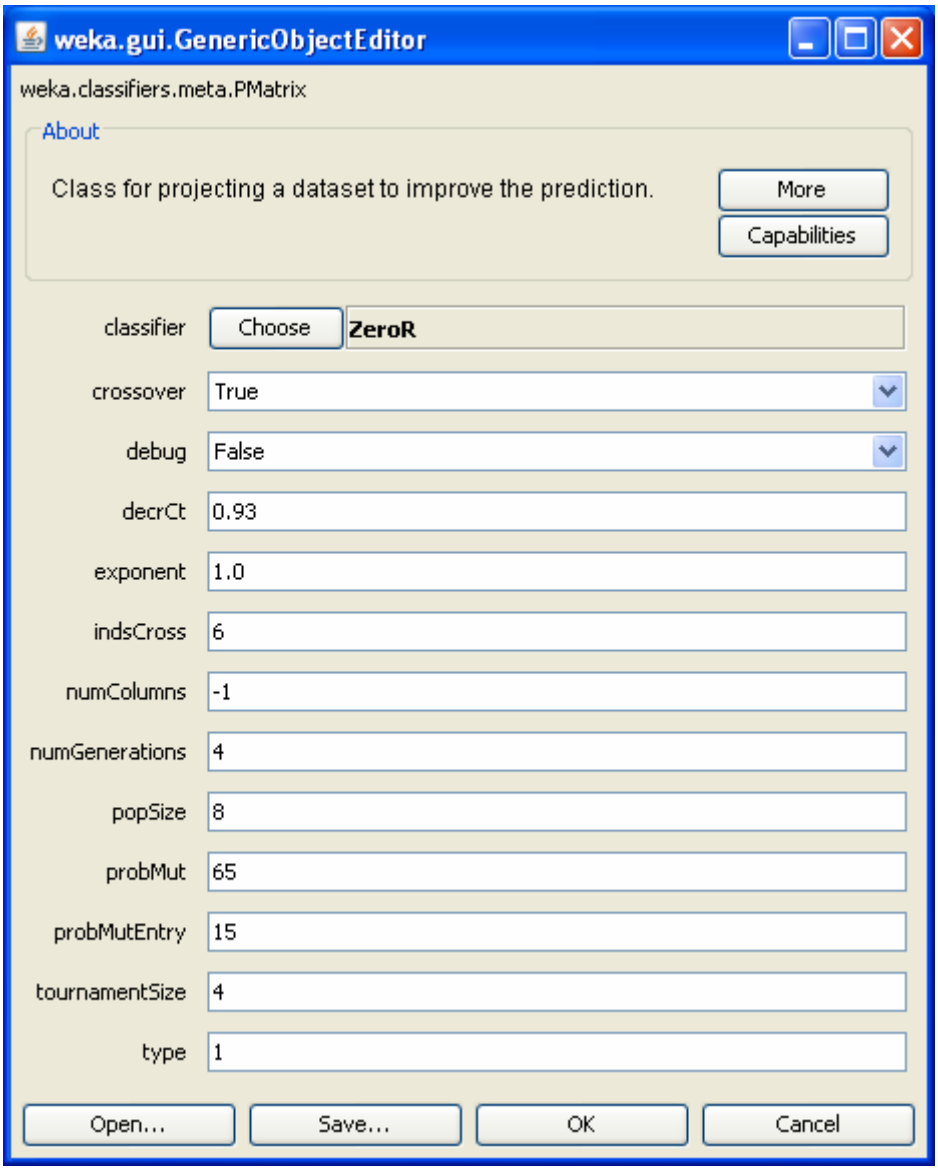

Ilustración B.5 – Ventana de configuración de parámetros de *PMatrix*

Con *PMatrix* puede aprenderse con cualquiera de los cuatro modos de test disponibles en Weka.

En el desarrollo de *PMatrix* se barajaron dos alternativas:

1. No modificar el núcleo de Weka y tener en cuenta simples suposiciones a la hora de ejecutar el algoritmo.
2. Modificar el núcleo de Weka[4] y simplificar ligeramente el uso del meta-algoritmo.

Sin embargo, puesto que Weka es software libre orientado al desarrollo y las suposiciones eran mínimas, se escogió la primera de ellas. Las suposiciones que han de tenerse en cuenta y los motivos son los siguientes:

[4] Esta opción implica que *PMatrix* no pueda ser añadido directamente a Weka como el resto de algoritmos.

- En primer lugar, desde el botón *More options…* debe estar siempre seleccionada la opción *Output model*.

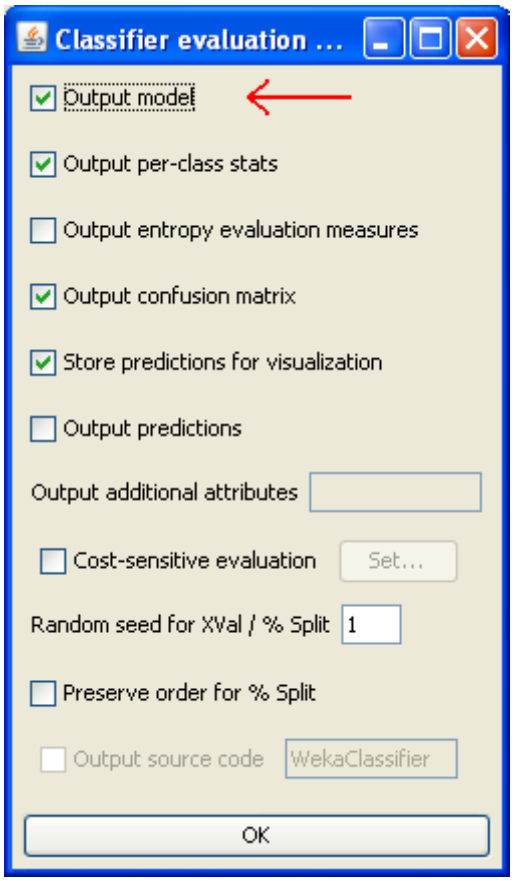

Ilustración B.6 – Ventana con opciones de ejecución

Esta opción es necesaria ya que es la única manera de que *PMatrix* sepa cuál es el conjunto de datos completo del cual se desea aprender, algo necesario en ciertas ocasiones.

- En segundo lugar, el método principal del algoritmo, *buildClassifier()*, puede ser llamado en más de una ocasión desde el núcleo de Weka y, puesto que es necesario discernir cuál de ellas es la primera vez que se llama en cada ejecución, se utiliza el método *getOptions()*.

  Este método tiene el inconveniente de que no sólo es invocado al principio de cada ejecución, sino que también es llamado cada vez que la ventana se mueve o se pierde su foco, por tanto, al usuario se le avisa al comienzo de la ejecución con el siguiente mensaje:

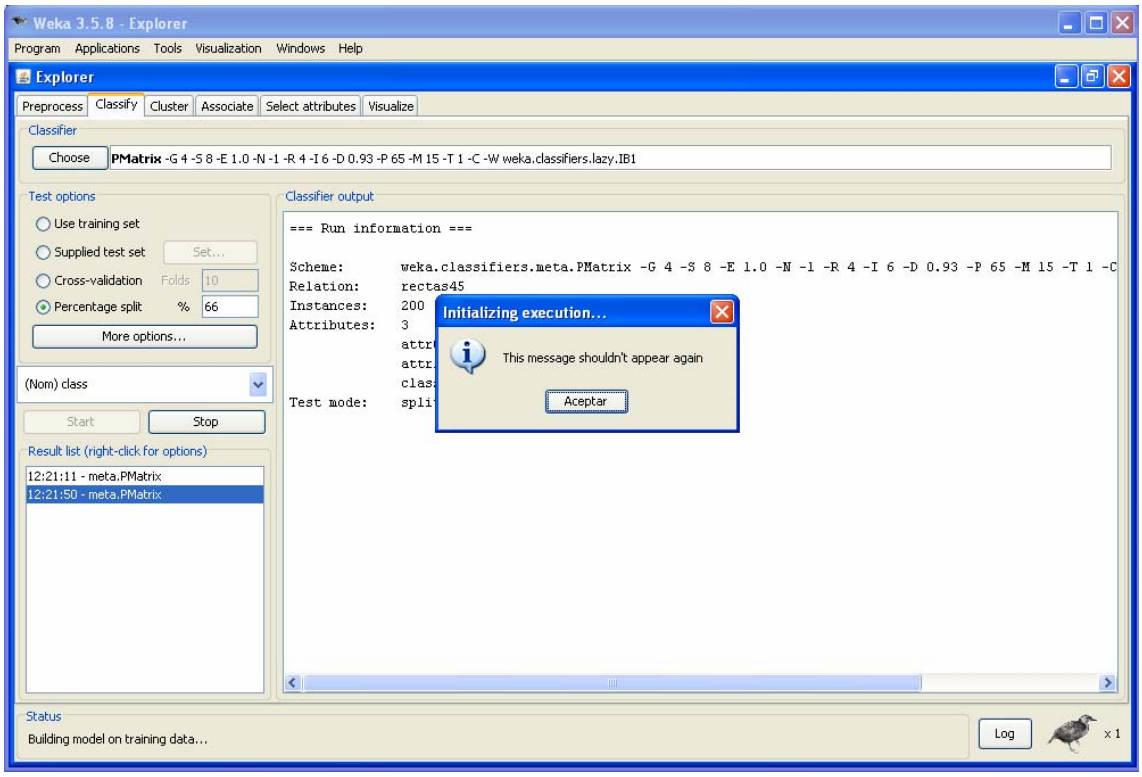

Ilustración B.7 – Mensaje de advertencia al comienzo de cada ejecución

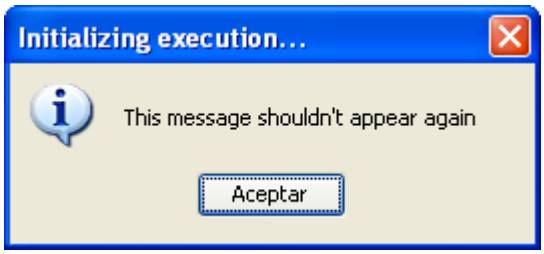

Ilustración B.8 – Mensaje de advertencia al comienzo de cada ejecución

Como se indica en las ilustraciones B.7 y B.8, este mensaje no debería aparecer de nuevo, ya que se consideraría una nueva ejecución lo que, en la mayoría de las ocasiones, provocaría un error antes de ser mostrado de nuevo. El error que aparecería se muestra en la ilustración B.9:

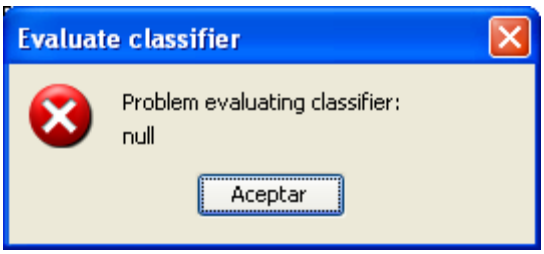

Ilustración B.9 – Mensaje de error

Si la ejecución del meta-algoritmo finaliza correctamente, el usuario dispone de los ficheros de salida de la ejecución en el directorio donde se encontraba el ejecutable[5].

B.3.2 Uso desde la consola

El algoritmo *PMatrix* puede ser usado también desde consola. Para acceder a ella debe pulsarse en la opción *SimpleCLI* de la pestaña *Applications* de la ventana principal de Weka.

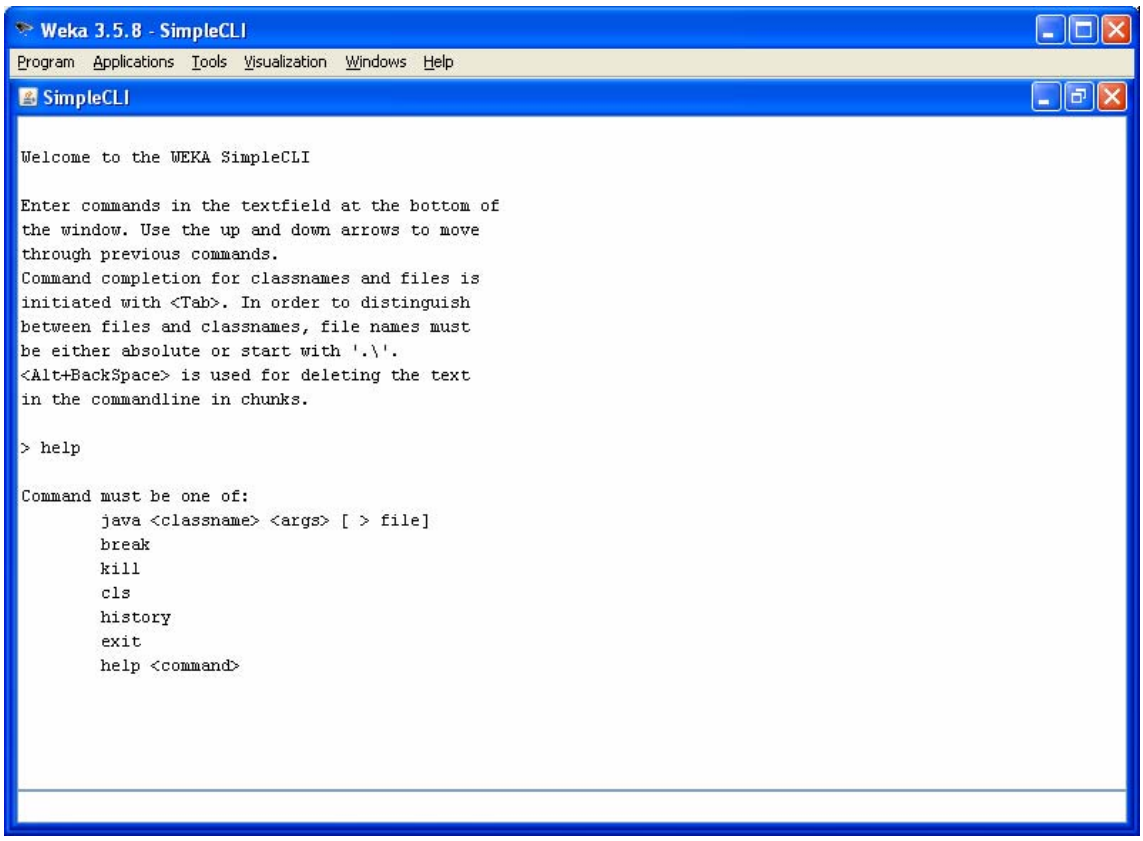

Ilustración B.10 – Ventana principal de la consola de *Weka*

Una vez en la consola, podría ejecutarse el meta-algoritmo con el siguiente comando:

java weka.classifiers.meta.PMatrix -W weka.classifiers.trees.J48 –no-cv -t diabetes.arff

Este comando ejecuta *PMatrix* con el clasificador base J48 sobre el fichero *diabetes.arff*, sin realizar validación cruzada y tomando el resto de parámetros sus valores por defecto.

[5] La estructura y contenido de los ficheros de salida de *PMatrix* están detallados en el capítulo 6.

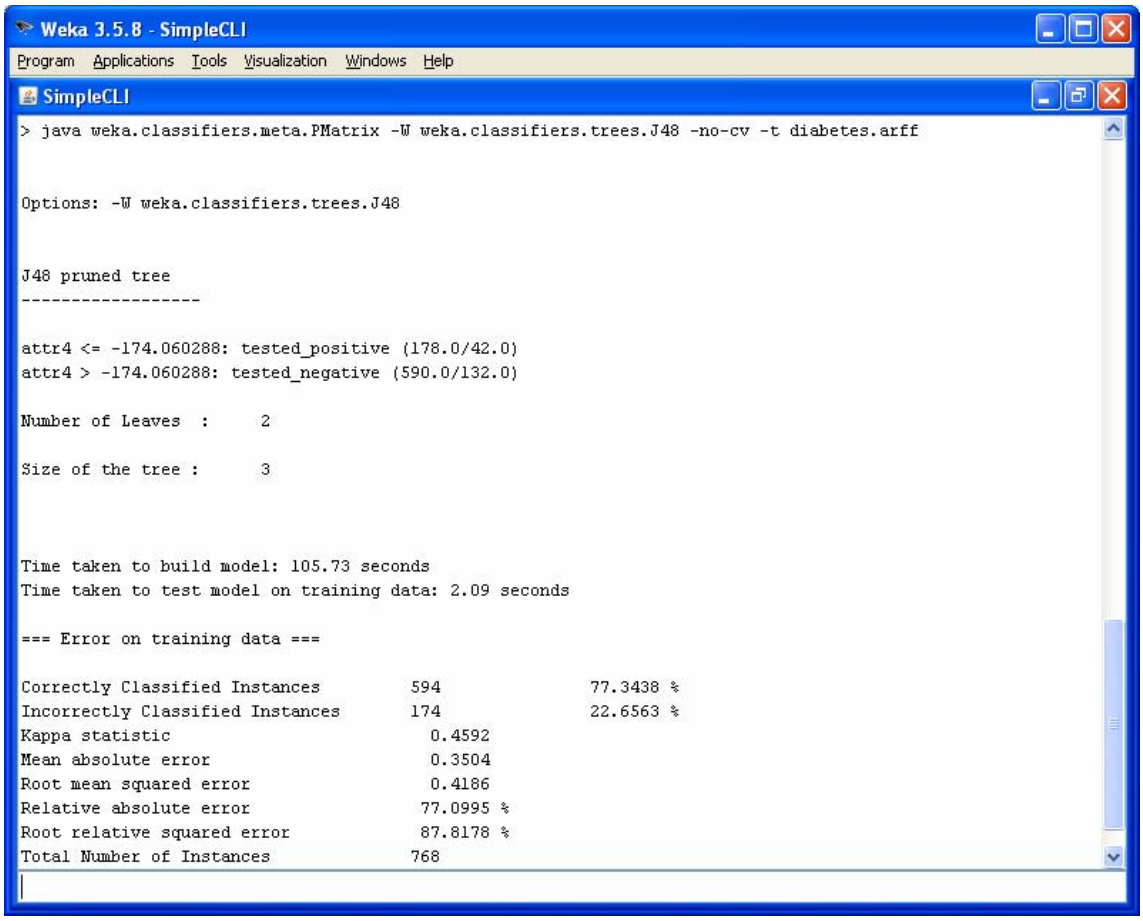

Ilustración B.11 – Resultados por consola

Puede observarse la salida generada por el comando anterior en la ilustración B.11. Pueden introducirse comandos más complejos en el que se detalla el valor de cada uno de los parámetros de *PMatrix*, por ejemplo:

weka.classifiers.meta.PMatrix -G 5 -S 8 -E 2.0 -N -1 -R 4 -I 6 -D 0.03 -P 75 -M 19 -T 2 -C -W weka.classifiers.rules.ZeroR –t diabetes.arff

B.3.3 Configuración avanzada

El meta-algoritmo desarrollado dispone de múltiples parámetros de entrada que pueden ser configurados por el usuario. A continuación se describe detalladamente cada uno de ellos:

- **classifier**: Permite seleccionar qué clasificador base utilizará el meta-algoritmo *PMatrix* para evaluar las proyecciones que realice. Puede ser seleccionado cualquier algoritmo implementado en Weka. El algoritmo base por defecto es ZeroR.
- **crossover**: Permite indicar al usuario si desea que se realice cruce. Si el usuario activa el cruce se lleva a cabo en cada generación una selección por torneos y el

posterior cruce de los individuos seleccionados; por el contrario, si el cruce está desactivado la población es mutada. La opción de cruce está habilitada por defecto.

- **debug**: Permite activar o desactivar la depuración. Si la depuración está activada se generan dos ficheros de texto adicionales en cada ejecución: debug.txt y debugP.txt[6]. La opción de depuración está deshabilitada por defecto.
- **decrCt**: Constante de decremento de la varianza de los individuos cada generación sin mejora. En computación evolutiva se supone que si la población no mejora se debe a que se está cerca de la solución, por tanto, al decrementarse la varianza se explora más en la zona del espacio de búsqueda en la que estén los individuos. El rango de valores posibles está en el intervalo [0.03, 0.97], siendo 0.93 el valor por defecto.
- **exponent**: Exponente al cual se eleva cada una de los valores de los atributos proyectados. El rango de valores posibles está en el intervalo (0, +∞), siendo 1.0 el valor por defecto.
- **indsCross**: Número de individuos nuevos que se generan en el cruce. El rango de valores posibles está en el intervalo [3, +∞), siendo 6 el valor por defecto. El número de nuevos individuos no tiene límite superior, ya que la política de reemplazamiento consiste en incorporarlos a la población sustituyendo al individuo con peor fitness que hubiera en ese momento, pudiendo darse el caso de que los nuevos individuos reemplacen a otros nuevos individuos recién incorporados. Este es el motivo por el cual el número de nuevos individuos puede ser mayor que el tamaño de la población.
- **numColumns**: Número de columnas de la matriz que proyectará los datos. El rango de valores posibles está en el intervalo {-1} U [1, +∞), siendo -1 el valor por defecto. Cuando el valor es -1 el número de columnas es fijado al número de atributos del conjunto de datos de entrada. También es fijado a este valor cuando el tipo de la matriz es 2 ó 3 (ver parámetro *type*).
- **numGenerations**: Máximo número de generaciones que el algoritmo seguirá buscando la solución sin que haya habido mejoras de fitness. El rango de valores posibles está en el intervalo [3, +∞), siendo 6 el valor por defecto.
- **popSize**: Tamaño de la población de individuos. El rango de valores posibles está en el intervalo [5, +∞), siendo 15 el valor por defecto.
- **probMut**: Probabilidad de mutación de cada individuo. Cuando el cruce está desactivado la población es mutada cada generación, siendo *probMut* la probabilidad de que cada individuo sea mutado en esa generación. El rango de valores posibles está en el intervalo [0, 100], siendo 50 el valor por defecto.
- **probMutEntry**: Probabilidad de mutación de cada entrada de la matriz. Es decir, si un determinado individuo va a ser mutado en una determinada generación, cada entrada de su matriz es mutada con probabilidad

[6] El contenido de ambos ficheros está explicado en el capítulo 6.

probMutEntry. El rango de valores posibles está en el intervalo [0, 100], siendo 15 el valor por defecto.

- **tournamentSize**: Tamaño de los torneos a la hora de seleccionar individuos para el cruce. El rango de valores posibles está en el intervalo [3, +∞), siendo 3 el valor por defecto. Si dicho valor supera el tamaño de la población, el tamaño de los torneos es reducido a dicho valor.
- **type**: Permite especificar el tipo de individuo (matriz) a utilizar al realizar las proyecciones. Sus posibles valores son {1, 2, 3} correspondiendo 1 con una matriz completa, 2 con una matriz simétrica y 3 con una matriz diagonal. Cuando la matriz es simétrica o diagonal, el número de columnas es fijado a -1. El valor por defecto de dicho parámetro es 1.

Cuando alguno de los parámetros toma un valor inferior al valor mínimo permitido, el valor del parámetro es establecido al valor mínimo, ídem para el valor máximo. Por el contrario, si toma un valor no numérico, el parámetro es establecido a su valor por defecto.

B.3.4 Acceso a la ayuda

El meta-algoritmo dispone de ayuda a la que puede accederse desde el explorador y desde consola.

Desde la ventana de parámetros del meta-algoritmo en el explorador, puede consultarse qué es cada atributo situando el cursor sobre el campo de texto en el que aparece el valor del atributo, la ayuda que se obtiene para el parámetro *crossover* puede verse en la ilustración B.12.

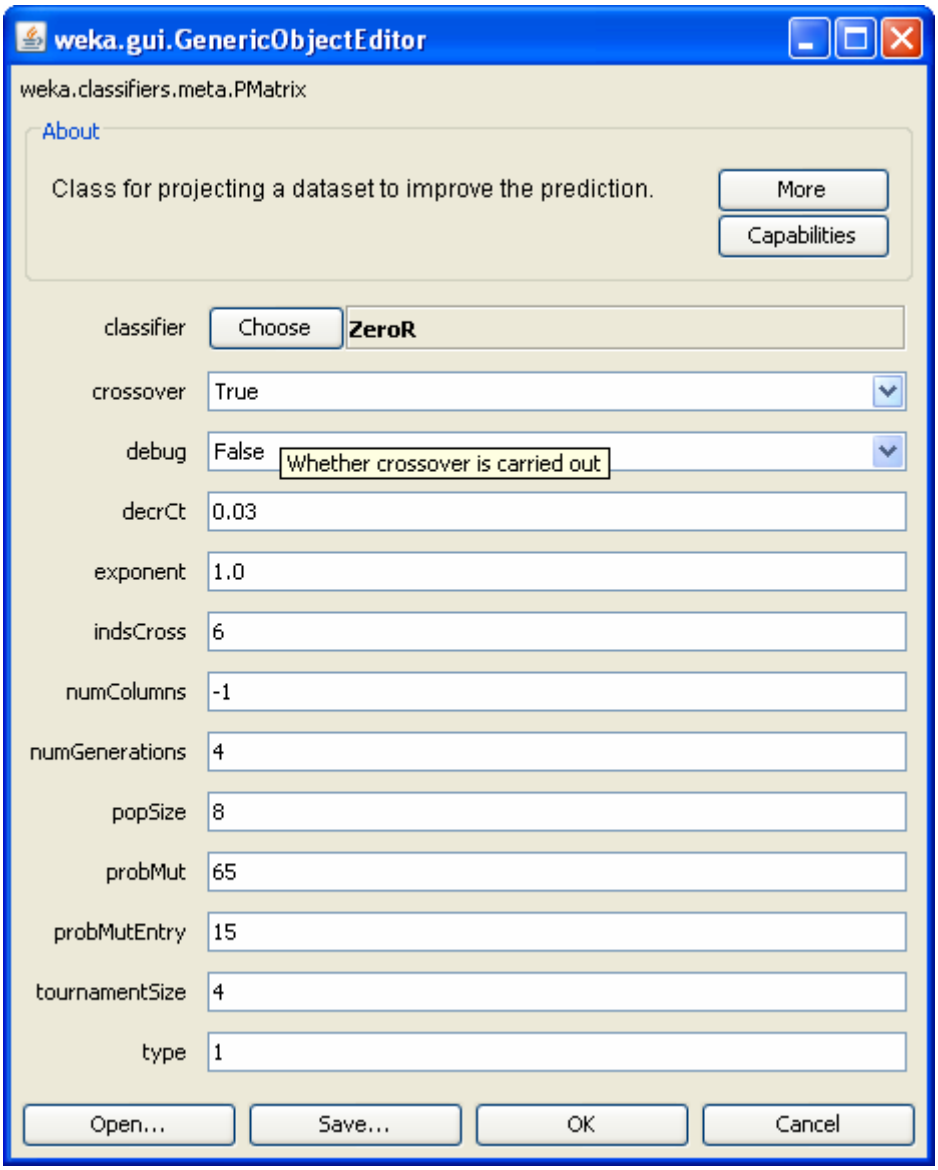

Ilustración B.12 – Ayuda para cada parámetro desde el explorador

Además, pulsando el botón *More* se abre una nueva ventana indicando información general sobre el meta-algoritmo y cada uno de sus parámetros.

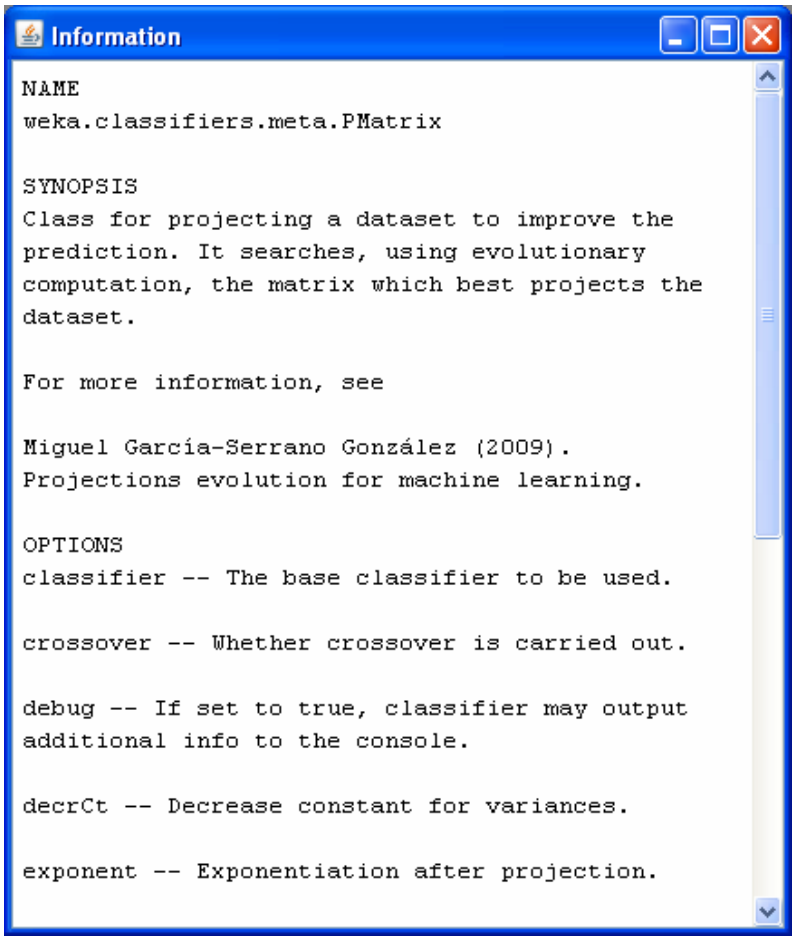

Ilustración B.13 – Ventana de ayuda desde el explorador

El comando para acceder a la ayuda del meta-algoritmo desde consola es el siguiente:

java weka.classifiers.meta.PMatrix

Con dicho comando aparece una lista de opciones generales y específicas para dicho algoritmo.

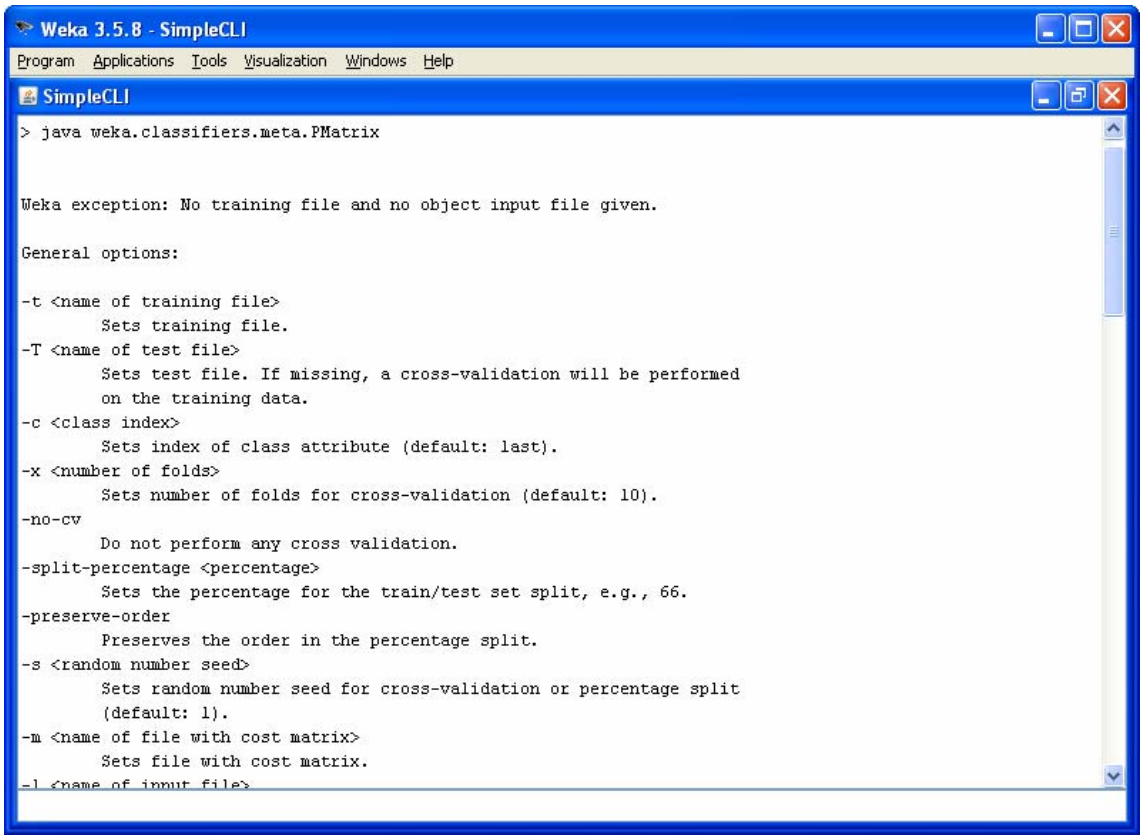

Ilustración B.14 – Opciones generales del algoritmo *PMatrix*

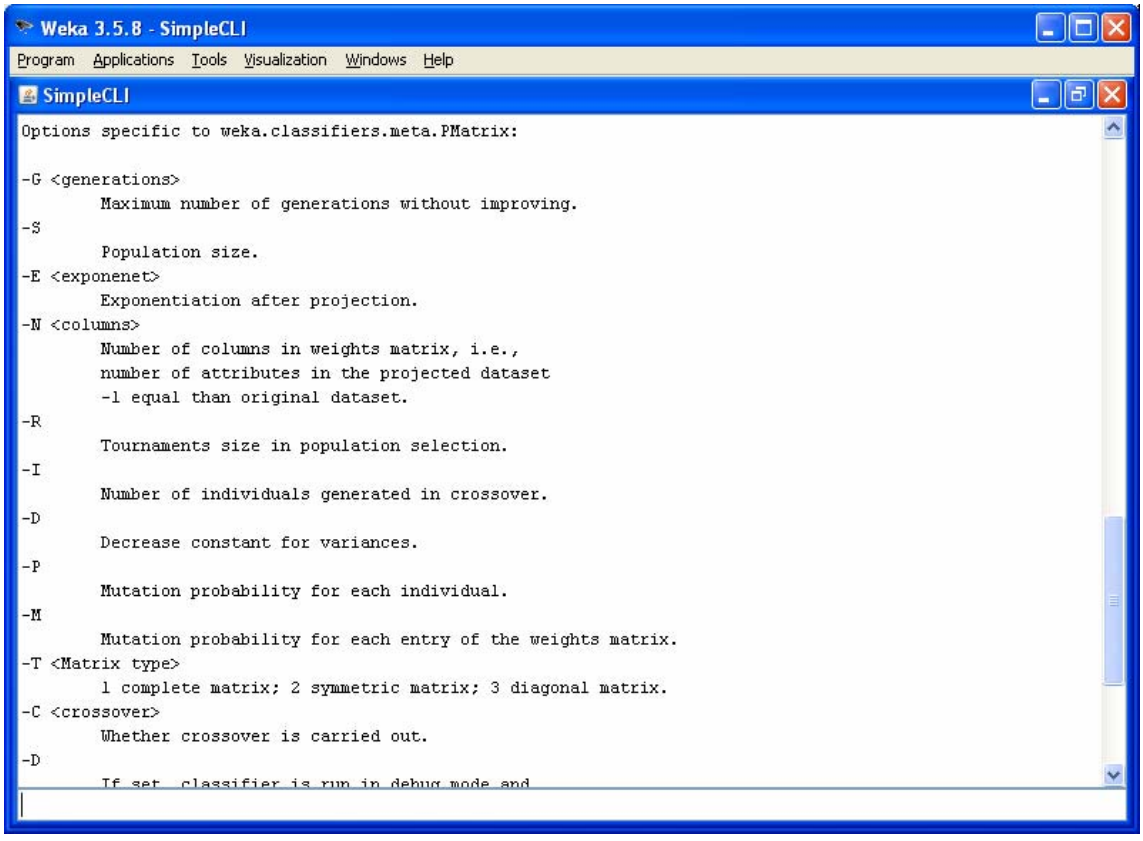

Ilustración B.15 – Opciones específicas del algoritmo *PMatrix*

B.4 Información técnica

El algoritmo ha sido desarrollado utilizando el lenguaje JAVA, necesario para la integración en *Weka*, en concreto, *PMatrix* ha sido integrado en la versión de desarrollador 3.5.8.

Durante el desarrollo del algoritmo ha sido utilizado el entorno de desarrollo de software libre *Eclipse*, en su versión 3.3.0, configurado con *JDK* 1.6.

La compilación con *Eclipse* se realizará con el uso de *Ant*, de manera que se genere un fichero con extensión *jar* con que puede ejecutarse y probar directamente el algoritmo desarrollado.

El meta-algoritmo consta de cuatro clases. En el paquete *weka.classifiers.meta* se encuentra la clase PMatrix.java y en el paquete *weka.classifiers.meta.pmatrix* se encuentran las restantes clases: Auxiliar.java, MotorGen.java e Individuo.java.

El meta-algoritmo genera como salida archivos ascii txt y arff.

El tamaño total del fichero ejecutable (*weka.jar*) es de 5240 KB.

C. Bibliografía

[1] Ian H. Witten y Eibe Frank. *Data Mining: Practical machine learning tools and techniques*, 2nd Edition, Morgan Kaufmann, San Francisco, 2005.

[2] Darwin, C. (1865). *The Origin of Species by Means of Natural Selection*, 6th ed. London: John Murray.

[3] J. H. Holland, *Concerning efficient adaptive systems*, in M. C. Yovits, G. T. Jacobi and G. D. Goldstein, eds., *Self-Organizing Systems—1962*, Spartan Books, Washington D.C., 1962, pp. 215 - 230.

[4] J. H. Holland, *Outline for a logical theory of adaptive systems*, Journal of the Association for Computing Machinery, 9 (1962), pp. 297 - 314.

[5] John H. Holland. *Adaptation in Natural and Artificial Systems*: 2nd edition. MIT Press. 1992

[6] Cramer, Nichael Lynn (1985), *A representation for the Adaptive Generation of Simple Sequential Programs* in Proceedings of an International Conference on Genetic Algorithms and the Applications, Grefenstette, John J. (ed.), Carnegie Mellon University.

[7] Koza, J.R. (1990), *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*, Stanford University Computer Science Department technical report.

[8] Pearson, K. (1901). *On Lines and Planes of Closest Fit to Systems of Points in Space*. Philosophical Magazine **2** (6): 559–572.

[9] Gilad-Bachrach, Ran. *Dimensionality Reduction for Online Learning Algorithms using Random Projections*. School of Computer Science and Engineering. The Hebrew University, Jerusalem, Israel. ranb@cs.huji.ac.il

[10] Wu, Mingrui. *Local Linear Projections*. Max Planck Institute for Biological Cybernetics, Tübingen, Germany. mingrui.wu@tuebingen.mpg.de

[11] Yu-En Lu, Pietro Li'o and Steven Hand. *On Low Dimensional Random Projections and Similarity Search*. Computer Laboratory. University of Cambridge

[12] Andreas Buja and George W. Furnas (1993). *Prosection Views: Dimensional Inference through Sections and Projections*. Bellcore.

[13] Marvin Minsky and Seymour Papert, 1972 (2nd edition with corrections, first edition 1969) *Perceptrons: An Introduction to Computational Geometry*, The MIT Press, Cambridge MA, ISBN: 0 262 63022 2.

[14] W. D. Cannon, *The Wisdom of the Body*, Norton and Company, New York, 1932.

[15] Turing, A. M., 1948, *Intelligent machinery*.
Reimpreso en: 1992, Mechanical Intelligence: Collected Works of A. M. Turing, D. C. Ince, ed., North-Holland, Amsterdam, pp. 107–127.
También reimpreso en: 1969, Machine Intelligence 5, B. Meltzer, and D. Michie, ed., Edinburgh University Press, Edinburgh.

[16] José M. Valls, Ricardo Aler y Óscar Fernández. *Evolving Generalized Euclidean Distances For Training RBNN*. Universidad Carlos III de Madrid. Madrid, 2006.

[17] A. M. Turing, *Computing Machinery and Intelligence*, Mind, 59 (1950), pp. 94-101.

[18] Weinberger, K. Q., Saul, L. K. 2009. *Distance Metric Learning for Large Margin Nearest Neighbor Classification*. Journal of Machine Learning Research, 10, 207-244.

[19] A. S. Fraser, *Simulation of Genetic Systems by Automatic Digital Computers I. Introduction*, Australian Journal of Biological Sciences, 10 (1957), pp. 484 - 491.

[20] A. S. Fraser, *Simulation of Genetic Systems by Automatic Digital Computers II. Effects of Linkage on Rates of Advance Under Selection.*, Australian Journal of Biological Sciences, 10 (1957), pp. 150-162.

[21] A. S. Fraser, *Simulation of Genetic Systems by Automatic Digital Computers VI. Epistasis*, Australian Journal of Biological Sciences, 13 (1960), pp. 150 - 162.

[22] H. J. Bremermann, *The evolution of intelligence. The nervous system as a model of its environment*, University of Washington, Seattle, 1958.

[23] R. M. Friedberg, *A Learning Machine: Part I*, IBM Journal of Research and Development, 2 (1958), pp. 2 - 13.

[24] G. J. Friedman, *Digital simulation of an evolutionary process*, General Systems: Yearbook of the Society for General Systems Research, 4 (1959), pp. 171 - 184.

[25] G. J. Friedman, *Selective Feedback Computers for Engineering Synthesis and Nervous System Analogy*, University of California, Los Angeles, 1956.